МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное
образовательное учреждение
высшего профессионального образования
«КРАСНОЯРСКИЙ ГОСУДАРСТВЕННЫЙ ПЕДАГОГИЧЕСКИЙ
УНИВЕРСИТЕТ им. В.П. АСТАФЬЕВА»
федеральное государственное бюджетное
образовательное учреждение
высшего профессионального образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

М.П. Варыгина

ОСНОВЫ ПРОГРАММИРОВАНИЯ В CUDA

Учебное пособие

Красноярск 2012

ББК 32.97

В 188

Рецензенты

В.М. Садовский,
доктор физико-математических наук, профессор

Н.И. Пак,
доктор педагогических наук, профессор

В 188 Варыгина М.П. Основы программирования в CUDA: учебное пособие / Краснояр. гос. пед. ун-т им. В.П. Астафьева. – Красноярск, 2012. – 138 с.

Разработано на основе материалов курса, внедренного в институте математики Сибирского федерального университета. Соответствует требованиям государственного стандарта по направлению Педагогическое образование, профиль «информатика».

Разработано при поддержке программы стратегического развития КГПУ им. В.П. Астафьева. Проект № 03-1/12.

ББК 32.97

© Красноярский государственный
педагогический университет
им. В.П. Астафьева, 2012
© Сибирский федеральный
университет, 2012
© Варыгина М.П., 2012

Содержание

Введение **5**

1 Модель программирования **10**
1.1 Основные понятия 10
1.2 Архитектура GPU 12
1.3 Расширение языка C 18
1.4 Основы работы с CUDA API 25
Вопросы и задания 37

2 Иерархия памяти **39**
2.1 Виды памяти 39
2.2 Работа с глобальной памятью 41
2.3 Работа с константной памятью 43
2.4 Работа с разделяемой памятью 45
Вопросы и задания 47

3 Оптимизация работы с памятью **49**
3.1 Оптимизация работы с глобальной памятью 49
3.2 Оптимизация работы с разделяемой памятью 57
Вопросы и задания 61

4 Реализация базовых операций **63**
4.1 Параллельная редукция 63
4.2 Транспонирование матрицы 75
4.3 Умножение матрицы на вектор 78
4.4 Умножение матриц 82
4.5 Решение трехдиагональных систем уравнений 87
4.6 Конечно-разностные методы 93
Вопросы и задания 94

5 Оптимизация приложений **95**
5.1 Математические функции 95
5.2 Использование потоков 96
5.3 Занятость мультипроцессора 99

5.4 Использование CUDA-профайлера 101
5.5 Использование отладчиков и PTX-ассемблера 104
Вопросы и задания 106

6 Работа с несколькими устройствами 107
6.1 Получение информации об имеющихся GPU 107
6.2 Выбор устройства 108
6.3 Потоки и события 109
6.4 Единое адресное пространство 110
6.5 Копирование с устройства на устройство 112
6.6 Декомпозиция области 113
6.7 Использование OpenMP 115
6.8 Использование MPI 115
Вопросы и задания 116

7 Библиотеки CUDA 117
7.1 CUFFT 117
7.2 CUBLAS 120
7.3 CUSPARSE 124
7.4 CURAND 130
7.5 Потоки 133
Вопросы и задания 133

Темы проектов 135

Библиографический список 137

Введение

В последнее время активно развивается отрасль высокопроизводительных вычислений. Для эффективного решения сложных наукоемких задач традиционной вычислительной архитектуры на основе центрального процессора теперь недостаточно. Возрастающий спрос на высокую вычислительную производительность обусловливает появление новых технологий. Одной из таких технологий является гибридная вычислительная модель, когда совместно с центральным процессором задача решается на графическом процессоре.

Графическое процессорное устройство (GPU, Graphic Processor Unit, Рис. 1) рассматривается как специализированное вычислительное устройство, которое является сопроцессором для центрального процессора (CPU, Central Processing Unit), обладает собственной памятью и возможностью параллельной обработки множества потоков исполнения.

Рис. 1. Графическое процессорное устройство NVIDIA Tesla

Рост производительности графического процессора значительно опережает рост производительности CPU (Рис. 2). GPU также обеспечивает

меньшее потребление энергии по сравнению с традиционными CPU кластерами.

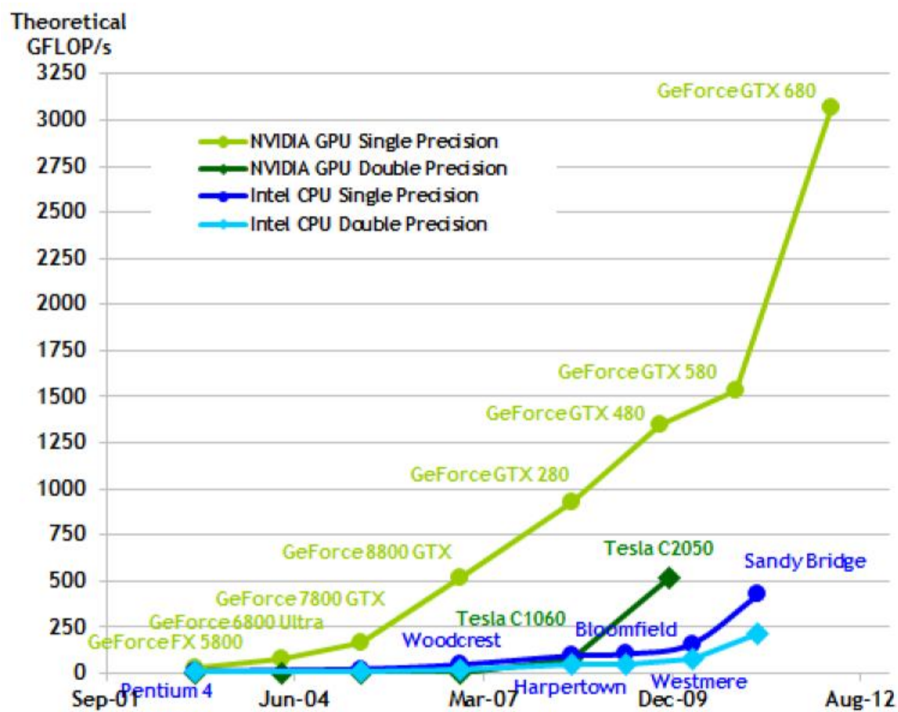

Рис. 2. Динамика роста частоты CPU и GPU

Графическое процессорное устройство впервые изобретено в компании NVIDIA (GeForce 256) в августе 1999 года для ускорения вывода трехмерной графики. Попытки использовать GPU для неграфических вычислений велись с 2003 года. С использованием шейдерных языков высокого уровня, таких как DirectX, OpenGL и Cg, различные параллельные алгоритмы переносились на GPU. Высокая производительность вычислений с плавающей точкой графических процессоров значительно ускоряла работу научных приложений. Это стало началом мощного дви-

жения, называющегося GPGPU, или вычислениями общего назначения на GPU.

Однако у подхода GPGPU имелись недостатки. Во-первых, от программиста требовалось знание графических API и архитектуры GPU. Для GPGPU были необходимы графические языки программирования, такие как OpenGL. Во-вторых, задачи должны были быть выражены в терминах координат вершин, текстур и пикселей, что увеличивало сложность программы. Разработчикам приходилось делать научные проекты похожими на графические приложения, отрисовывая треугольники и полигоны, а также реализовывать алгоритмы на специальном шейдерном языке. В-третьих, основные особенности программирования, такие как произвольный доступ к памяти на чтение и запись, не поддерживались, возникали ограничения на программную модель. И наконец, недостаток поддержки двойной точности (до недавнего времени) означал, что многие научные приложения не могли исполняться на GPU.

Для решения этих проблем NVIDIA представила две технологии: архитектуру G80 (впервые в GeForce 8800, Quadro FX 5600 и Tesla C870) и *CUDA* (Compute Unified Device Architecture) – программно-аппаратную архитектуру для вычислений на GPU. CUDA – универсальная архитектура параллельных вычислений, которая дает возможность организации доступа к набору инструкций графического ускорителя и управления его памятью при организации параллельных вычислений. Вместе эти технологии представляли новый способ использования GPU. Вместо программирования графических APIs программист может писать программы с CUDA-расширениями. Для разработки приложений с использованием

архитектуры CUDA может использоваться целый набор языков и API, включая C, C++, Fortran, Java, Python, OpenCL и Direct Compute.

Область научных задач, решаемых с помощью технологии CUDA, очень широка. Вычисления на GPU сейчас применяются в научных приложениях вычислительной математики, механики жидкостей и газов, сейсмической разведки, астрономии и астрофизики, анализа и обработки изображений, обработки звука и видео, финансовой математики, криптографии, искусственного интеллекта, геоинформационных систем, биоинформатики, магнитно-резонансной томографии, молекулярной динамики.

Данное учебное пособие посвящено основам программирования графических процессоров по технологии CUDA. В нем вводятся основные понятия, используемые для описания организации параллельных вычислений на GPU. Представлены особенности архитектуры графического устройства и API, знание которых необходимо для разработки эффективных параллельных алгоритмов.

В первой главе дается общее введение в программирование на графических устройствах, рассматривается их архитектура, описывается синтаксис CUDA, представляющий собой расширение языка C. Также приводятся примеры простейших программ, описывается процесс установки CUDA на компьютер и компиляции программ.

Вторая и третья главы посвящены работе с памятью на устройстве. Во второй главе рассматриваются основные виды памяти, доступные на устройстве, а в третьей – приводятся способы оптимизации доступа к памяти.

В четвертой главе приводятся примеры основных базовых операций линейной алгебры, таких как параллельная редукция, транспонирование матрицы, умножение матрицы на вектор, умножение матриц, а также некоторые способы организации программы при использовании конечно-разностных методов.

Пятая глава посвящена повышению производительности работы программы, здесь рассматриваются некоторые способы оптимизации приложений.

В шестой главе описывается возможность использования нескольких устройств в одной программе.

В седьмой главе рассматриваются библиотеки CUDA: CUFFT – для расчета быстрого преобразования Фурье; CUBLAS – для решения задач линейной алгебры; CUSPARSE – для работы с разреженными векторами и матрицами, и CURAND – для генерации случайных чисел.

1 Модель программирования

1.1 Основные понятия

Программный код в CUDA состоит как из последовательных, так и из параллельных частей (Рис. 3). Последовательные части кода выполняются на центральном процессоре. Массивно-параллельные части кода выполняются на GPU как функция–*ядро* (kernel function).

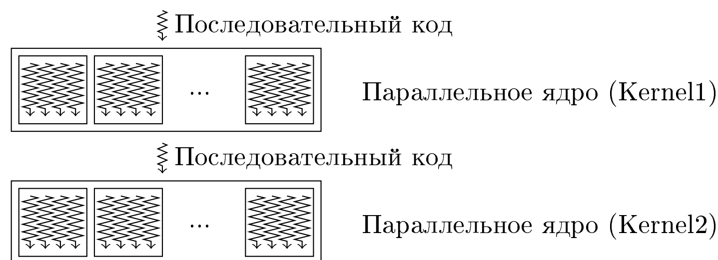

Рис. 3. Гетерогенная структура кода

Параллельная часть кода выполняется как большое число *нитей* (threads). При этом обычно каждой нити соответствует один элемент вычисляемых данных. Все запущенные на выполнение нити организованы в следующую иерархию (Рис. 4). Нити группируются в *блоки* (blocks). Каждый блок – это одномерный, двумерный или трехмерный массив нитей. Блоки объединяются в *сеть блоков* (grid) – верхний уровень иерархии. Сетка блоков представляет собой одномерный или двумерный массив блоков. При этом все блоки, образующие сетку, имеют одинаковую размерность и размер.

Подобное разделение всех нитей является еще одним общим приемом использования CUDA – исходная задача разбивается на набор отдельных

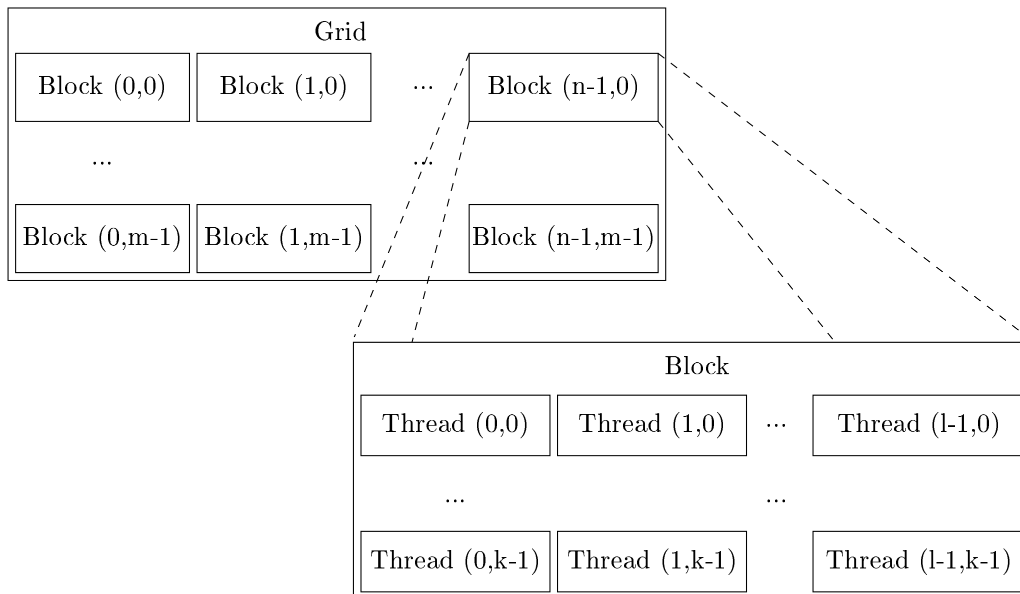

Рис. 4. Иерархия нитей в CUDA

подзадач, решаемых независимо друг от друга (Рис. 5). Каждой подзадаче соответствует блок нитей. При этом каждая подзадача совместно решается всеми нитями своего блока, и нити могут взаимодействовать между собой в пределах блока.

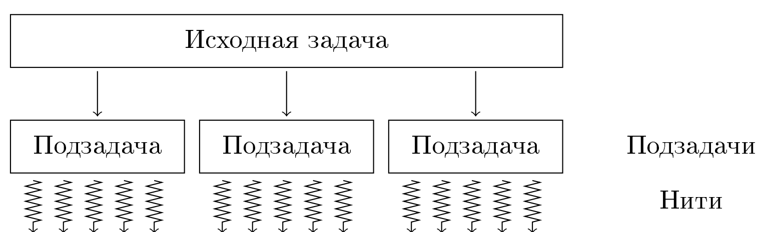

Рис. 5. Разбиение исходной задачи на набор подзадач

Подобный подход похож на работу с SIMD-моделью (Single Instruction – Multiple Data, одна инструкция – множество данных), однако в

CUDA используется модель *SIMT* (Single Instruction – Multiple Thread, одна инструкция – множество нитей). Нити разбиваются на группы по 32 нити – *warp'ы*. Только нити в пределах одного warp'а выполняются физически одновременно. Нити из разных warp'ов могут находиться на разных стадиях выполнения программы. Управление warp'ами осуществляется GPU.

1.2 Архитектура GPU

GPU построен как масштабируемый массив *потоковых мультипроцессоров* (SM, Streaming Multiprocessor). Когда на CUDA запускается ядро на выполнение, то блоки сетки выполняются на имеющихся мультипроцессорах. При этом каждый блок целиком выполняется на одном из мультипроцессоров, который, в свою очередь, способен одновременно выполнять до восьми блоков. По мере того как отдельные блоки завершают свое выполнение, на их место становятся новые блоки. Поэтому даже на довольно небольшом числе потоковых мультипроцессоров можно запустить на выполнение сетку с большим числом блоков.

Многопоточная программа распределяется по блокам нитей, которые выполняются независимо друг от друга, поэтому GPU с большим числом ядер выполнит программу за меньшее время, чем GPU с меньшим числом ядер (Рис. 6).

Для обозначения возможностей GPU CUDA использует понятие *Compute Capability* (cc), выражаемое парой чисел – major.minor. Первое число обозначает глобальную архитектурную версию, второе – небольшие изменения в архитектуре.

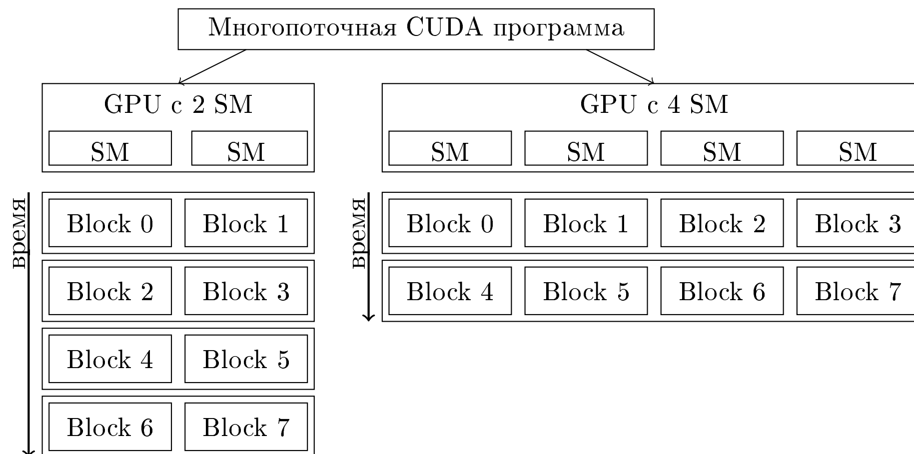

Рис. 6. Масштабируемая программная модель

Для написания оптимальных программ на графических процессорах важно знать особенности архитектуры устройства. Рассмотрим архитектуру графических процессоров серии G80, Tesla 10 и Fermi.

1.2.1 Архитектура G80

На рисунке 7 схематически представлена архитектура графического процессора для GPU GeForce 8800. Видеокарта содержит 8 кластеров текстурных блоков (TPC, Texture Processing Cluster), каждый из которых состоит из двух мультипроцессоров и одного текстурного блока для работы с текстурной памятью. Таким образом, на видеокарте находятся 16 потоковых мультипроцессоров, каждый из которых содержит 8 *скалярных ядер* (SP, Scalar Processor). Каждая нить выполняется на одном из скалярных ядер.

Кроме скалярных ядер, потоковый мультипроцессор содержит также 2 специальных функциональных блока (SFU, Special Function Unit) для

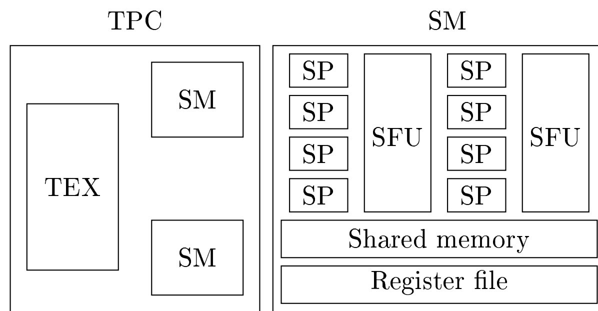

Рис. 7. Схема TPC и потокового процессора G80

вычисления некоторых отдельных функций, блок управления командами (Instruction Unit) и свою память.

Мультипроцессор содержит внутри себя память следующих типов:

- набор 32-битовых регистров (register file);
- разделяемая память, доступная всем ядрам;
- кэш константной памяти, доступный на чтение всем ядрам;
- кэш текстурной памяти, доступный на чтение всем ядрам.

Мультипроцессор управляет созданием, выполнением и уничтожением нитей, группируя их в warp'ы по 32 нити. При этом именно warp является единицей управления – для каждого warp'а отслеживается готовность данных для его выполнения. На каждом шаге выбирается warp, для которого готовы все данные, и выполняется одна команда для всех составляющих его нитей. После этого определяется что нужно данному warp'у для продолжения, выбирается следующий готовый warp и так далее.

1.2.2 Архитектура Tesla

На рисунке 8 представлена архитектура графического процессора серии Tesla10.Видеокарта содержит 10 текстурных блоков, каждый из которых имеет 3 потоковых мультипроцессора по 8 скалярных процессоров. Таким образом, на видеокарте содержится 240 скалярных процессоров.

Потоковый мультипроцессор этой серии содержит также специальный блок для обработки 64-битовых чисел с плавающей точкой (Double Precision Unit).

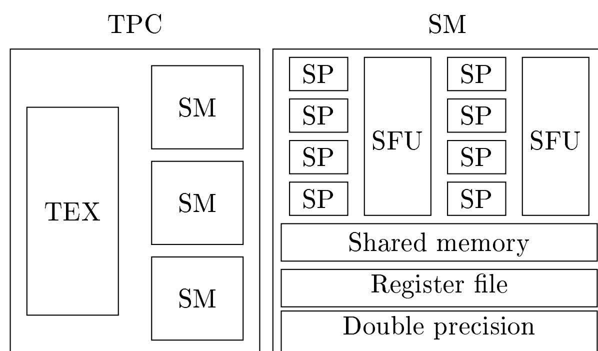

Рис. 8. Схема TPC и потокового процессора Tesla10

1.2.3 Архитектура Fermi

Графический процессор, основанный на архитектуре Fermi (Рис. 9), содержит 512 скалярных ядер, которые организованы в 16 потоковых мультипроцессоров, расположенных вокруг общего L2 кэша. В каждом мультипроцессоре содержится по 32 ядра. На устройстве содержатся 6 64-битных модулей памяти (до 6GB GDDR5). Host Interface связывает

GPU с CPU через шину PCI-Express. GigaThread Global Scheduler распределяет блоки нитей по планировщикам нитей на мультипроцессорах.

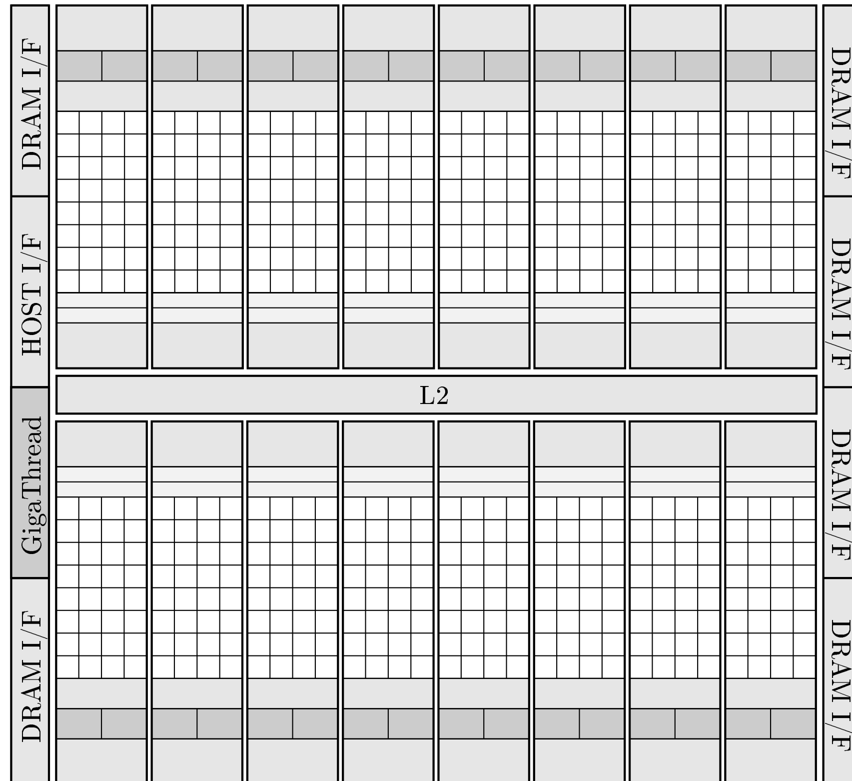

Рис. 9. Архитектура Fermi

Каждый мультипроцессор (Рис. 10) имеет 16 блоков загрузки/записи, которые позволяют вычислять адреса источника и места назначения для 16 нитей за один такт. На каждом мультипроцессоре есть 4 блока SFU для вычисления специальных функций, которые выполняют математические инструкции, такие как sin, cos, sqrt. Каждый SFU выполняет одну инструкцию для нити за один такт.

Instruction Cache			
Scheduler		Scheduler	
Dispatch		Dispatch	
Register File			
Core	Core	Core	Core
Core	Core	Core	Core
Core	Core	Core	Core
Core	Core	Core	Core
Core	Core	Core	Core
Core	Core	Core	Core
Core	Core	Core	Core
Core	Core	Core	Core
Load/Store Units x16			
Special Func Units x 4			
Interconnect Network			
64K Configurable Cache/Shared Memory			

Рис. 10. Схема потокового мультипроцессора Fermi

Мультипроцессор распределяет нити в группы по 32 нити – warp'ы. У каждого мультипроцессора есть 2 планировщика warp'ов и 2 блока управления командами (Instruction Dispatch Unit), что позволяет двум warp'ам запускаться и выполняться одновременно.

Каждый мультипроцессор Fermi имеет 64 KB памяти, которая может быть распределена между разделяемой памятью и L1 кэшем: 48 KB для разделяемой памяти и 16 KB для L1 кэша, или 16 KB для разделяемой памяти и 48 KB для L1 кэша.

1.3 Расширение языка C

Вводимые в CUDA расширения языка C состоят из:

- спецификаторов функций, определяющих откуда вызывать и где выполнять функцию:
  `__device__`, `__global__` и `__host__`.
- спецификаторов переменных, задающих тип памяти, используемый для данных переменных:
  `__device__`, `__shared__` и `__constant__`.
- встроенных векторных типов;
- встроенных переменных для задания размера сетки / блока и индексов блока/нити;
- директив вызова ядра, задающих иерархию нитей:
  `kernel<<<GridDim,BlockDim>>>()`.

1.3.1 Спецификаторы функций

В CUDA используются следующие спецификаторы функций (Табл. 1).

Таблица 1. Спецификаторы функций в CUDA

Спецификатор	Функция выполняется на	Функция вызывается из
`__device__`	device (GPU)	device (GPU)
`__global__`	device (GPU)	host (CPU)
`__host__`	host (CPU)	host (CPU)

Спецификатор `__global__` обозначает ядро, и соответствующая функция должна возвращать значение типа `void`. При вызове ядра нужно обязательно указывать конфигурацию вызова.

На функции, выполняемые на GPU (спецификаторы `__device__` и `__global__`), накладываются следующие ограничения:

- не поддерживается рекурсия;
- не поддерживаются static-переменные внутри функции;
- не поддерживается переменное количество входных аргументов.

Адрес функции `__device__` нельзя использовать при указании на функцию (а `__global__` можно).

Спецификаторы `__host__` и `__device__` могут быть использованы вместе. Это означает, что соответствующая функция может выполняться как на GPU, так и на CPU.

Спецификаторы `__global__` и `__host__` не могут быть использованы вместе.

1.3.2 Спецификаторы переменных

Для размещения в памяти GPU переменных используются следующие спецификаторы: `__device__`, `__shared__` и `__constant__`. В таблице 2 приводятся основные характеристики добавленных переменных.

Таблица 2. Спецификаторы переменных

Спецификатор	Область действия	Срок жизни	Память устройства
`__device__`	сетка	приложение	глобальная
`__shared__`	блок	блок	разделяемая
`__constant__`	сетка	приложение	константная

На использование добавленных переменных накладываются следующие ограничения:

- соответствующие переменные могут использоваться только в пределах одного файла, их нельзя объявлять как `extern`;
- спецификаторы не могут быть применены к полям структуры;
- запись в переменные типа `__constant__` может осуществляться только CPU при помощи специальных функций;
- `__shared__` переменные не могут инициализироваться при объявлении.

1.3.3 Встроенные векторные типы

Добавлены 1 / 2 / 3 / 4-мерные векторы из базовых типов:

```
char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4;
short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4;
int1, uint1, int2, uint2, int3, uint3, int4, uint4;
long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4;
float1, float2, float3, float4;
double1, double2.
```

Обращение к компонентам вектора идет по именам – `x`, `y`, `z` и `w`.

```
uint4 param;
int y = param.y;
```

Для этих векторов не определены покомпонентные операции, то есть нельзя просто сложить два вектора при помощи оператора сложения – это необходимо явно делать для каждой компоненты.

Для создания значений векторов заданного типа служит конструкция вида make_<typename>.

```
int2 a = make_int2(1,7); // создать вектор (1,7)
```

Также добавлен тип dim3, используемый для задания размерности. Этот тип основан на типе uint3, но обладает при этом конструктором, инициализирующим все незаданные компоненты единицами.

```
dim3 blocks (16,16);
```

1.3.4 Встроенные переменные

Для того чтобы ядро могло однозначно определить номер нити (а значит, и элемент данных, который нужно обработать), используются встроенные переменные threadIdx и blockIdx типа dim3. Каждая из этих переменных является трехмерным целочисленным вектором.

Также ядро может получить размеры сетки и блока через встроенные переменные gridDim и blockDim типа dim3.

На рисунке 11 представлено отображение локальных номеров нити и блока в глобальный индекс нити.

Знная номер нити внутри блока, номер блока внутри сетки и размерность сетки, можно получить глобальный индекс нити:

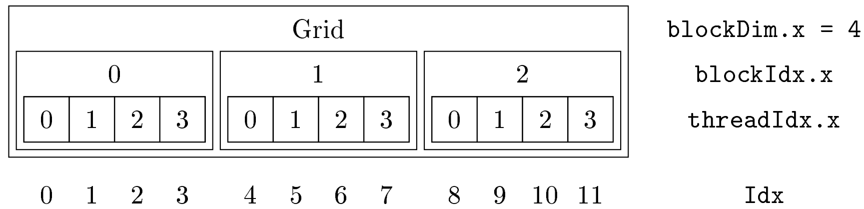

Рис. 11. Отображение локальных индексов массива в глобальные

```
Idx = blockIdx.x * blockDim.x + threadIdx.x
```

Для получения глобального индекса нити для двумерных и трехмерных массивов можно найти сначала индекс нити для каждого из направлений:

```
Idx = blockIdx.x * blockDim.x + threadIdx.x
Idy = blockIdx.y * blockDim.y + threadIdx.y
```

Для двумерного массива размера (`Dx, Dy`) глобальный индекс нити вычисляется как (`Idx + Idy * Dx`) (Рис. 12).

Аналогично для трехмерного массива размера (`Dx, Dy, Dz`) глобальный индекс нити равен (`Idx + Idy * Dx+ Idz * Dx* Dy`).

Максимальное число нитей в блоке ограничено, так как все нити блока располагаются на одном потоковом процессоре и должны разделять ограниченное число ресурсов этого процессора. Блок нитей может содержать до 512 или 1024 нитей (в зависимости от архитектуры). Однако ядро может быть запущено на большом числе блоков. Поэтому общее число нитей равно произведению числа нитей в блоке и числа блоков.

			0			1			2			blockId.x
			0	1	2	0	1	2	0	1	2	threadId.x
			0	1	2	3	4	5	6	7	8	Idx = blockIdx.x*blockDim.x+threadIdx.x
0	0	0					4					
	1	1			11							11=(0*3+2)+(0*3+1)*9
	2	2							24			24=(2*3+0)+(0*3+2)*9
1	0	3				30						30=(1*3+0)+(1*3+0)*9
	1	4										
	2	5										
2	0	6										
	1	7		64								
	2	8									80	80=(2*3+2)+(2*3+2)*9

Idy

threadIdx.y

blockIdx.y

Id=Idx+Idy*9

Рис. 12. Вычисление глобального индекса двумерного массива

1.3.5 Директива вызова ядра

Для запуска ядра на GPU в простейшем случае используется следующая конструкция:

```
kernel<<<dG,dB>>>(args);
```

Здесь `kernel` – это имя (адрес) соответствующей `__global__` функции. Через `dG` обозначена переменная (или значение) типа `dim3`, задающая размерность и размер сетки (в блоках). Переменная (или значение) `dB` типа `dim3` задает размерность и размер блока (в нитях).

Размерности сетки (в блоках) и блока (в нитях) могут быть заданы несколькими способами:

```
dim3 grid, block;
grid.x = 2; grid.y = 4;
block.x = 8; block.y = 16;
```

или

```
dim3 grid(2, 4);
dim3 block(8,16);
```

Через `args` обозначены аргументы вызова функции `kernel` (их может быть несколько).

Следующий код запускает ядро с именем `kernel` параллельно на n нитях, используя одномерный массив из двумерных (16×16) блоков нитей, и передает ядру два параметра – a и n.

```
kernel<<<dim3(n/256), dim3(16,16)>>>(a,n);
```

Ниже приводится фрагмент кода программы, увеличивающей элементы массива на единицу, в реализации для CPU и GPU.

CPU:
```
float * Data;
for (int i=0; i<n; i++){
    Data[i] = Data[i] + 1.0f;
}
```

GPU:
```
 __global__ void incKernel(float * Data){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    Data[i] = Data[i] + 1.0f;
}
```

1.4 Основы работы с CUDA API

Технология CUDA предоставляет в распоряжение программиста ряд функций, которые могут быть использованы только CPU (CUDA host API). CUDA API для CPU выступает в двух формах: низкого уровня (CUDA driver API) и высокого уровня (CUDA runtime API, реализованный через CUDA driver API). В программе можно использовать только один из них.

Низкоуровневый CUDA driver API дает больше возможностей программисту, а также требует большего объема кода, явных настроек, явной инициализации.

Высокоуровневый CUDA runtime API не требует явной инициализации – она происходит автоматически при первом вызове какой-либо его функции. В дальнейшем будем использовать именно CUDA runtime API.

1.4.1 Установка CUDA на компьютер

Для установки CUDA на компьютер необходимо установить:

1. Драйвер видеокарты NVIDIA с поддержкой CUDA;
2. CUDA Toolkit, CUDA Tools;

3. CUDA SDK – примеры программных проектов.

Для компиляции программ на CUDA потребуется установленный компилятор с C/C++. В качестве такого компилятора в Microsoft Windows может выступать компилятор cl, входящий в состав Microsoft Visual Studio, а также компилятор cygwin, в Linux – компилятор gcc.

1.4.2 Компиляция и запуск программ

Для компиляции программ на CUDA существует компилятор `nvcc`, использующий внешний компилятор для компиляции частей кода, выполняемых на CPU. Функции, составляющие ядро, помещаются в файл с расширением `.cu`, который компилируется с использованием программы `nvcc`.

Для того чтобы просто откомпилировать программу, состоящую из одного или нескольких файлов, сразу в выполняемый файл, можно воспользоваться следующей командой (для Microsoft Windows):

```
nvcc file1.cu file2.cu file3.cpp -o program.exe
```

Для Linux команда выглядит аналогично, только расширение `.exe` для выполняемого файла не указывается:

```
nvcc file1.cu file2.cu file3.cpp -o program
```

Для сборки проектов, состоящих из многих файлов, можно также воспользоваться утилитой `make` (или ее аналогом в Microsoft Windows – утилитой `nmake`).

Если на компьютере установлен SDK, то можно воспользоваться шаблоном типовой программы. Для этого необходимо:

- скопировать папку template из NVIDIA_GPU_Computing_SDK / C / src / template в свой каталог;
- откомпилировать проект командой `make`;
- запустить файл на исполнение.

В результате выполнения программы должно появиться сообщение «PASSED» (Рис. 13):

```
cuda@ubuntu:~/NVIDIA_GPU_Computing_SDK/C/src/template$ make
cuda@ubuntu:~/NVIDIA_GPU_Computing_SDK/C/src/template$ ./template
[template] starting...
Processing time: 64.773003 (ms)
[template] test results...
PASSED

Press ENTER to exit...
```

Рис. 13. Компиляция и запуск программы

Содержимое Makefile:

```
# Add source files here
EXECUTABLE  := template
# CUDA source files (compiled with cudacc)
CUFILES     := template.cu
# CUDA dependency files
```

```
CU_DEPS      := \
    template_kernel.cu \

# C/C++ source files (compiled with gcc / c++)
CCFILES      := \
    template_gold.cpp \

# Rules and targets
include ../../common/common.mk
```

В секции EXECUTABLE указывают имя выходного файла, CUFILES – имя файла с CUDA-программой. В CU_DEPS обычно указывают файлы, где хранятся функции-ядра или вспомогательные __device__ функции. В CCFILES указывают имена файлов с расширениями .c, .cpp.

1.4.3 Работа с памятью

Память на устройстве выделяется и освобождается CPU при помощи следующих вызовов:

```
//выделение памяти
cudaMalloc(void ** devPtr, size_t size);
//освобождение памяти
cudaFree(void ** devPtr);
```

Для копирования данных с CPU на GPU и обратно используется следующий вызов:

```
//копирование данных
cudaMemcpy(void * dst, const void * src,
           size_t size, enum cudaMemcpyKind kind);
```

В качестве значения параметра `kind` выступает одна из следующих констант, задающих направление копирования:

- `cudamemcpyHostToDevice`;
- `cudamemcpyDeviceToHost`;
- `cudamemcpyDeviceToDevice`;
- `cudamemcpyHostToHost`.

Простой пример работы функций выделения, копирования и освобождения памяти приведен ниже.

```
float *h_a, *h_b;  // host
float *d_a, *d_b;  // device
int n = 1024;
int data_sz = n*sizeof(float);
 //Выделение памяти на CPU:
h_a = (float *)malloc(data_sz);
h_b = (float *)malloc(data_sz);
 //Выделение памяти на GPU:
cudaMalloc((void **) &d_a, data_sz);
cudaMalloc((void **) &d_b, data_sz);
 //Инициализация массива на CPU:
for (int i=0, i<n; i++) h_a[i] = 10.f + i;
 //Копирование массива a на GPU:
cudaMemcpy(d_a, h_a, data_sz, cudaMemcpyHostToDevice);
 //Копирование массива b на GPU:
cudaMemcpy(d_b, d_a, data_sz, cudaMemcpyDeviceToDevice);
 //Копирование массива b на CPU:
cudaMemcpy(h_b, d_b, data_sz, cudaMemcpyDeviceToHost);
 //Сравнение результатов:
for (int i=0; i<n; i++) assert(h_a[i] == h_b[i]);
 //Освобождение памяти на CPU и GPU:
free(h_a);
```

```
free(h_b);
cudaFree(d_a);
cudaFree(d_b);
```

1.4.4 Пример. Сложение векторов

Рассмотрим пример сложения двух векторов на GPU.

```
__global__ void sumKernel(float * a, float * b, float * c){
  // глобальный индекс нити
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  c[i] = a[i] + b[i];
}

void sum(float * a, float * b, float * c, int n){
    int data_sz = n * sizeof(float);
    float * d_a = NULL;
    float * d_b = NULL;
    float * d_c = NULL;
    // выделение памяти в GPU
    cudaMalloc((void**)&d_a,data_sz);
    cudaMalloc((void**)&d_b,data_sz);
    cudaMalloc((void**)&d_c,data_sz);
    //копирование данных из памяти CPU в память GPU
    cudaMemcpy(d_a,a,data_sz,cudaMemcpyHostToDevice);
    cudaMemcpy(d_b,b,data_sz,cudaMemcpyHostToDevice);
    //вызов ядра для обработки данных
    sumKernel <<<dim3(512),dim3(n/512)>>>(d_a,d_b,d_c);
    //копирование результатов в память CPU
    cudaMemcpy(c,d_c,data_sz,cudaMemcpyDeviceToHost);
    //освобождение памяти
    cudaFree(d_a);
```

```
    cudaFree(d_b);
    cudaFree(d_c);
}
```

Первая функция `sumKernel()` является ядром, выполняющим сложение векторов. Спецификатор `__global__` указывает на то, что функция вызывается из CPU и выполняется на GPU. Ядро при помощи встроенных переменных вычисляет глобальный индекс нити, для которого выполняется сложение, и суммирует соответствующие элементы.

Функция `sum()` выделяет память на GPU, копирует входные данные из памяти на CPU в память GPU, вызывает ядро `sumKernel()`, копирует результат обратно в память CPU и освобождает выделенную память.

1.4.5 Типичная структура CUDA программы

В приведенном выше примере хорошо показано использование CUDA. Типичная структура CUDA программы имеет вид:

1. Объявление глобальных переменных;
2. Прототипы функций:
   - `__global__ void kernel();`
3. `Main()`:
   - Размещение памяти на устройстве – `cudaMalloc();`
   - Передача данных с процессора на устройство – `cudaMemcpy();`
   - Определение параметров запуска ядра;
   - Вызов ядра – `kernel<<<block, thread>>>();`
   - Передача результатов с устройства на процессор – `cudaMemcpy();`

- Дополнительно: сравнение результатов расчета на процессоре и устройстве;

4. Ядро – `__global__ void kernel()`:

   - Объявление переменных – `local`, `shared`;
   - `__syncthreads()`.

1.4.6 Обработка ошибок

Каждая функция CUDA runtime API возвращает значение типа `cudaError_t`. При успешном выполнении функции возвращается значение `cudaSuccess`, в противном случае возвращается код ошибки. Получить описание ошибки в виде строки можно при помощи функции `cudaGetErrorString()`:

```
char * cudaGetErrorString(cudaError_t error);
```

Также можно получить код последней ошибки при помощи функции `cudaGetLastError()`:

```
cudaError_t cudaGetLastError();
```

Следующий макрос обеспечивает простой интерфейс для проверки вызовов CUDA:

```
#define cudaVerify(x) do {                                              \
  cudaError_t __cu_result = x;                                          \
```

```
  if (__cu_result!=cudaSuccess) {                                  \
    fprintf(stderr,"%s:%i: error: cuda function call failed:\n"\
        "  %s;\nmessage: %s\n",                                     \
        __FILE__,__LINE__,#x,cudaGetErrorString(__cu_result)); \
    exit(1);                                                          \
   }                                                                  \
  } while(0)
#define cudaVerifyKernel(x) do {                                     \
  x;                                                                  \
  cudaError_t __cu_result = cudaGetLastError();                      \
  if (__cu_result!=cudaSuccess) {                                    \
    fprintf(stderr,"%s:%i: error: cuda function call failed:\n"\
        "  %s;\nmessage: %s\n",                                       \
       __FILE__,__LINE__,#x,cudaGetErrorString(__cu_result));  \
    exit(1);                                                          \
   }                                                                  \
  } while(0)
```

Макрос `cudaVerify(x)` ожидает, что выражение `x` вернет ошибку `cudaError_t`. Если возникает ошибка, то макрос выводит на печать строку с описанием ошибки и завершает работу программы. Этот макрос подходит для большинства вызовов CUDA library.

Макрос `cudaVerifyKernel(x)` выполняет выражение `x` и вызывает функцию `cudaGetLastError()`. Если возникает ошибка, то макрос выводит на печать строку с описанием ошибки и завершает работу программы. Этот макрос подходит для вызовов ядер.

Замечание: вызов ядра должен быть окружен скобками для обработки препроцессором:

```
cudaVerifyKernel((kernel<<<...>>>(...)))
```

1.4.7 Замеры времени на GPU

Для точного измерения времени выполнения различных операций на GPU можно воспользоваться так называемыми *событиями* (CUDA events). Событие – это объект типа `cudaEvent_t`, используемый для обозначения «точки» среди вызовов CUDA. Каждое событие, привязанное к точке, характеризуется тем, пройдена GPU данная точка или нет.

CUDA runtime API обеспечивает точный замер времени, позволяя приложению асинхронно записывать события в любой точке программы, запрашивать наступило ли данное событие, ждать наступления события, а также получать интервал времени в миллисекундах между наступлениями событий.

Ниже приводится простой пример кода, замеряющий время выполнения ядра на GPU.

```
cudaEvent_t start, stop; //инициализируем события
float elapsedTime;
cudaEventCreate(&start);//создаем события
cudaEventCreate(&stop);

cudaEventRecord(start, 0);//записываем событие

//вызываем ядро

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);//ждем завершения работы ядра
```

```
cudaEventElapsedTime(&elapsedTime, start, stop);
cudaEventDestroy(start);//уничтожаем события
cudaEventDestroy(stop);
```

1.4.8 Синхронизация

Для синхронизации текущей нити на CPU с GPU используется функция `cudaThreadSynchronize()`, которая дожидается завершения выполнения всех операций CUDA, ранее вызванных с данной нити CPU.

Функция CPU `cudaDeviceSynchronize()` блокирует выполнение кода CPU пока все GPU ядра не выполнены.

Для синхронизации всех нитей в одном и том же блоке используется функция `__syncthreads()`, которая блокирует вызывающие нити до тех пор, пока все нити блока не войдут в эту функцию.

При помощи этой функции можно организовать барьеры внутри ядра, гарантирующие, что если хотя бы одна нить прошла такой барьер, то не осталось ни одной за барьером (не прошедшей его). Такая синхронизация называется *барьерной* (Рис. 14).

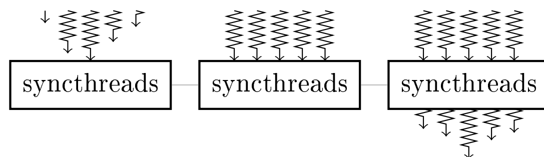

Рис. 14. Синхронизация нитей в блоке

1.4.9 Ветвления

При выполнении блока на потоковом мультипроцессоре все его нити перенумеровываются и разбиваются на группы по 32 нити – warp'ы (группа потоков, выполняемых физически параллельно). Все нити warp'а выполняют одну и ту же команду, при этом нити разных warp'ов могут выполнять разные команды.

Если нити одного warp'а должны идти по разным веткам кода (например, из-за условного оператора), то выполняются все проходимые ветки. Это называется *ветвлением* (divergence, branching), приводящим к снижению быстродействия. При этом нити, принадлежащие другим warp'ам, не оказывают никакого влияния на данный warp. Чем больше ветвлений внутри warp'а, тем медленнее он выполняется, из-за необходимости пройти все встречающиеся ветви кода. Старайтесь избегать ветвления, когда условия ветвления зависят от идентификатора нити.

Ниже представлены примеры с расходящимся ветвлением:

```
if (threadIdx.x == 0)
 {...}
else
 {...}
```

Для первой нити первого warp'а будет выполняться одна ветка, а для всех остальных нитей первого warp'а – другая, то есть имеется ветвление внутри warp'а. В результате все нити данного warp'а выполнят обе ветви, что приведет к снижению быстродействия ядра.

```
if (threadIdx.x > 2) { }
```

В этом примере номер нити меньше размера warp'a, что приведет к ветвлению.

```
if (threadIdx.x / WARP_SIZE > 2) { }
```

В данном примере номер нити кратен размеру warp'a. Ветвления внутри warp'a не произойдет.

Очень часто расходящиеся ветвления возникают в уравнениях в частных производных с граничными условиями:

- простые граничные условия: для реализации граничных условий можно организовать цикл по всем внутренним узлам и ветвление там, где необходимо;
- граничные условия с большим объемом вычислений: первое ядро выполняет граничные условия, второе ядро обрабатывает внутренние узлы.

Вопросы и задания

1. Что такое GPU?
2. Дайте определения основным понятиям: ядро, сетка, блок, нить, warp.
3. Как устроена архитектура GPU? Приведите примеры.
4. Чем отличаются потоковый мультипроцессор и скалярный процессор?

5. Какие спецификаторы функций можно использовать вместе?
6. Для чего используется спецификатор `__global__` ?
7. Как связаны локальные и глобальные индексы нитей и блоков?
8. Как можно инициализировать переменную типа dim3?
9. С помощью какой команды происходит компиляция программы?
10. Опишите параметры вызова ядра.
11. С помощью какой функции выделяется память на устройстве?
12. С помощью какой функции происходит передача данных на устройство?
13. Опишите структуру CUDA-программы.
14. Как происходит обработка ошибок в CUDA?
15. Что такое событие в CUDA?
16. Опишите способ замера времени работы CUDA-программы.
17. В каких случаях требуется синхронизация нитей?
18. Что такое ветвление? Как оно влияет на производительность работы программы?
19. Напишите код функции `main()`, реализующий сложение векторов на GPU, которая вызывает функцию `sum()` в примере.
20. Добавьте в код функции `sum()` функции обработки ошибок.
21. К предыдущей задаче добавьте в код функции `sum()` функции замера времени выполнения работы программы.

2 Иерархия памяти

2.1 Виды памяти

Память GPU можно разделить на DRAM и на память, размещенную непосредственно на GPU в потоковых мультипроцессорах (Рис. 15). В таблице 3 приводятся доступные виды памяти в CUDA и их основные характеристики.

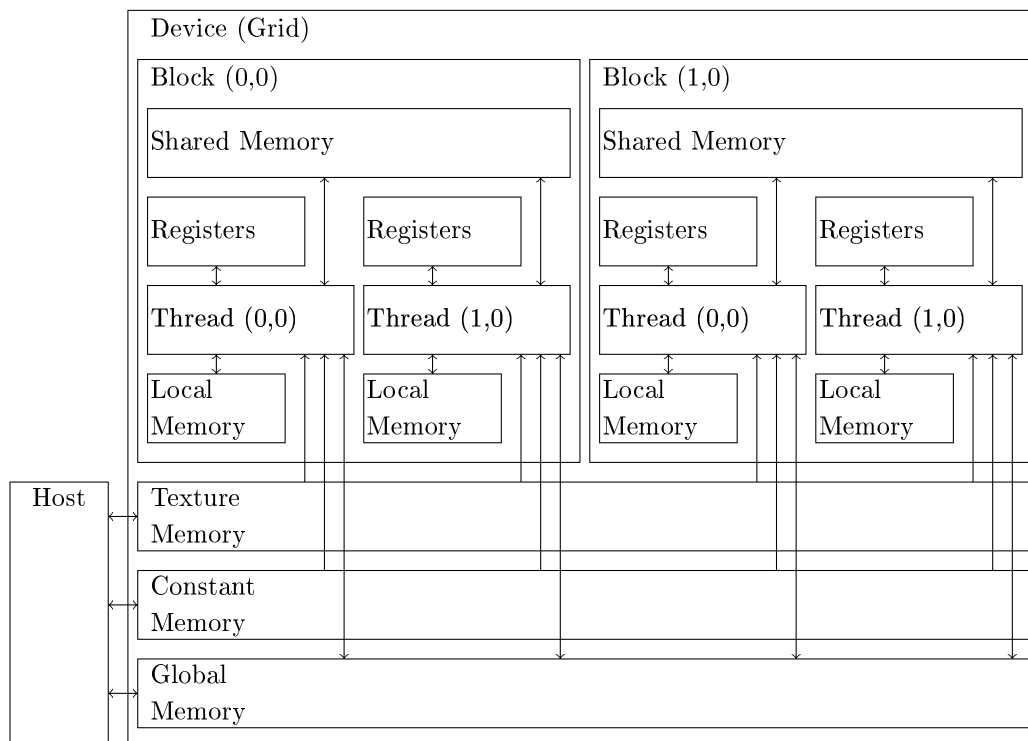

Рис. 15. Иерархия памяти в CUDA

Наиболее простым видом памяти является *регистровая память* (регистры, registers). Имеющиеся регистры распределяются между нитями

Таблица 3. Типы памяти в CUDA

Тип памяти	Расположение	Доступ	Доступ	Время жизни
Регистры	SM	r/w	per-thread	Нить
Локальная	DRAM	r/w	per-thread	Нить
Разделяемая	SM	r/w	per-block	Блок
Глобальная	DRAM	r/w	per-grid	Приложение
Константная	DRAM	r/o	per-grid	Приложение
Текстурная	DRAM	r/o	per-grid	Приложение

блока на этапе компиляции (и, соответственно, влияют на количество блоков, которые может выполнять один мультипроцессор). Вся работа по размещению данных в регистрах выполняется компилятором.

Каждая нить имеет доступ на чтение и запись некоторого количества регистров на протяжении выполнения данного ядра. Все переменные, определенные в регистрах, недоступны другим нитям. Поскольку регистры расположены непосредственно в потоковом мультипроцессоре, то они обладают максимальной скоростью доступа.

Если имеющихся регистров не хватает, то для размещения локальных данных (переменных) нити используется *локальная память* (local memory), размещенная в DRAM. Доступ к локальной памяти характеризуется очень высокой латентностью – от 400 до 600 тактов.

Переменные, объявленные в коде `__device__` функции без спецификаторов обычно размещаются в регистрах. В некоторых случаях компилятор может поместить эти переменные в локальную память.

Следующий вид памяти в CUDA – *разделяемая память* (shared memory). Она расположена непосредственно в потоковом мультипроцессоре, но выделяется на уровне блоков – каждый блок имеет доступ на чте-

ние и запись одного и того же количества разделяемой памяти. От того, сколько разделяемой памяти требуется одному блоку, зависит количество блоков, которое может быть запущено на одном мультипроцессоре. Доступ к разделяемой памяти может быть настолько же быстрым, как и доступ к регистрам.

Глобальная память (global memory) – это обычная DRAM-память, которая выделяется при помощи специальных функций на CPU. Все нити сетки имеют доступ на чтение и запись глобальной памяти. Глобальная память обладает высокой латентностью (от 400 до 600 тактов).

Константная память (constant memory) обычно используется для размещения небольшого объема часто используемых неизменяемых данных, которые должны быть доступны всем нитям сетки сразу.

Текстурная память (texture memory) предназначена главным образом для работы с текстурами. Она имеет специфические особенности в адресации, чтении и записи данных.

Константная память и текстурная память доступны сразу всем нитям сетки только на чтение. Запись в них может осуществляться CPU при помощи специальных функций.

Центральному процессору доступна лишь глобальная, константная и текстурная память.

2.2 Работа с глобальной памятью

Глобальная память является основным местом для размещения и хранения большого объема данных для обработки ядрами. Она сохраняет

свои значения между вызовами ядер, что позволяет через нее передавать данные между ядрами.

В разделе 1.4.3 были рассмотрены основные функции для работы с глобальной памятью. Функции cudaMalloc и cudaFree являются стандартными функциями выделения и освобождения глобальной памяти. Функция cudaMemset используется для записи значения в глобальную память.

```
//выделение памяти
cudaError_t cudaMalloc(void ** devPtr, size_t size);
//освобождение памяти
cudaError_t cudaFree(void ** devPtr);
//запись значения в память
cudaError_t cudaMemset(void * devPtr, int value, size_t size);
```

Функции выделения глобальной памяти возвращают указатель на память GPU. Доступ по данному указателю может осуществлять только код, выполняемый на GPU. Для доступа CPU к этой памяти следует воспользоваться функцией копирования памяти:

```
//копирование данных
cudaError_t cudaMemcpy(void * dst, const void * src,
                size_t size, enum cudaMemcpyKind kind);
```

Копирование памяти между CPU и GPU является дорогостоящей операцией, поэтому число подобных операций должно быть минимизировано. Это связано с тем, что скорость передачи данных ограничивается скоростью передачи данных по шине PCI Express.

Скорость передачи данных между CPU и GPU может быть повышена за счет использования pinned- (или page-locked) памяти.

```
//выделение pinned памяти
cudaError_t cudaMallocHost(void ** devPtr, size_t size);
//освобождение pinned памяти
cudaError_t cudaFreeHost(void ** devPtr);
```

Использование pinned-памяти имеет ряд преимуществ для некоторых устройств:

- копирование между pinned-памятью CPU и памятью устройства может осуществляться одновременно с выполнением ядра;
- pinned-память может быть отображена в память устройства, при этом нет необходимости копирования на устройство.

Однако такая память является ограниченным ресурсом, и ее использование в некоторых случаях может отрицательно сказаться на быстродействие всей программы.

2.3 Работа с константной памятью

Константная память выделяется непосредственно в коде программы при помощи спецификатора `__constant__`. Общий объем константной памяти ограничен – до 64 KB на устройстве. Все нити сетки могут читать из нее данные, и чтение из нее кешируется. CPU имеет доступ к ней как на чтение, так и на запись при помощи следующих функций:

```
cudaError_t cudaMemcpyToSymbol(const char * symbol, const void *
      src, size_t count, size_t offset, enum cudaMemcpyKind kind);
cudaError_t cudaMemcpyFromSymbol(void * dst, const char * symbol,
     size_t count, size_t offset, enum cudaMemcpyKind kind);
```

В качестве значения параметра `kind` выступает одна из следующих констант, задающих направление копирования:

- `cudamemcpyHostToHost`;
- `cudamemcpyDeviceToHost`;
- `cudamemcpyHostToDevice`;
- `cudamemcpyDeviceToDevice`.

Ниже приводится пример – фрагмент кода, объявляющий массив в константной памяти и инициализирующий этот массив данными из массива из памяти CPU.

```
__constant__ float constdata[256];
float hostdata [256];
// copy data from CPU to GPU constant memory
cudaMemcpyToSymbol("constdata", &hostdata,  sizeof(float),
      0,cudaMemcpyHostToDevice);
```

Замечание: в коде CPU переменные или массивы в константной памяти являются символами. Символ – идентификатор строки (`type = const char *`) в CUDA runtime API.

В Fermi константная память также используется для хранения аргументов ядра при вызове.

2.4 Работа с разделяемой памятью

Разделяемая память размещена непосредственно в самом мультипроцессоре и относится к быстрому типу памяти. Ее рекомендуется использовать для минимизации обращений к глобальной памяти, а так же для хранения локальных переменных функций. Адресация разделяемой памяти между нитями потока одинакова в пределах одного блока, что может быть использовано для обмена данными между потоками в пределах одного блока.

Для размещения данных в разделяемой памяти используется спецификатор `__shared__`.

Существует два способа управления выделением разделяемой памяти. Самый простой способ заключается в явном задании размеров массивов:

```
__shared__ float data[256];
```

При этом способе задания компилятор сам произведет выделение необходимого количества разделяемой памяти для каждого блока при запуске ядра.

Кроме того, можно также при запуске ядра задать дополнительный объем разделяемой памяти (в байтах), который необходимо выделить каждому блоку.

```
kernel <<<grid, block, k * sizeof(float)>>> (a);
//объявление внутри ядра
```

```
 __global__ void kernel (float * a){
    extern __shared__ float data[];
    ...
}
```

В приведенном выше примере кода каждому блоку будет дополнительно выделено `k * sizeof(float)` байт разделяемой памяти, доступной массиву `data`.

Если в блоке нитей больше одного warp'а, не всегда известно, в каком порядке warp'ы выполняют свои инструкции. Почти всегда требуется синхронизация нитей, чтобы обеспечить корректное использование разделяемой памяти. Инструкция `__syncthreads()` обеспечивает барьерную синхронизацию для всех нитей блока. Нить, достигшая барьера, ждет, пока все нити также не достигнут барьера.

Ниже приводится пример синхронизации разделяемой памяти. После того как каждая нить поместит данные в разделяемую память, требуется синхронизация – вызов `__syncthreads()`. После этого данные из разделяемой памяти могут быть использованы остальными нитями.

```
__global__ void kernel(){
  __shared__ float a[...];
  // нить 0 пишет a[0], нить 1 пишет a[1],...
  a[threadIdx.x] = ...;
  __syncthreads();
  // нить 0 читает a[1], нить 1 читает a[2],...
  ... = a[threadIdx.x+1];
}
```

При написании кода с условиями нужно быть внимательным, чтобы убедиться, что все нити достигнут вызова `__syncthreads()`, как пока-

зано в примере ниже.

```
if (...){
    ... // загрузка данных
}
__syncthreads();
if (...){
    ... // вычисления
}
__syncthreads();
```

Если вызов `__syncthreads()` произойдет внутри блока с условием, то может возникнуть ситуация, когда нити, дошедшие до барьера, будут ожидать оставшиеся нити, которые не войдут в этот блок.

Вопросы и задания

1. Какие виды памяти на GPU вы знаете?
2. Какие виды памяти доступны нитям только на чтение?
3. Какие виды памяти обладают высокой скоростью доступа к ним?
4. Какие виды памяти доступны CPU?
5. Опишите спецификаторы переменных.
6. Что такое регистры?
7. Для чего используется локальная память?
8. Для чего используется константная память?
9. Как организована работа с константной памятью?
10. Как организована работа с глобальной памятью?
11. Что такое pinned-память?

12. Как организована работа с разделяемой памятью?
13. Как можно разместить данные в разделяемой памяти?
14. Как организовать синхронизацию разделяемой памяти?

3 Оптимизация работы с памятью

Быстродействие программ на CUDA определяется двумя факторами – вычислительной производительностью (GFGlops/sec) и скоростью обращения к памяти (GBytes/sec). При этом чаще всего быстродействие ограничивается именно скоростью работы с памятью.

3.1 Оптимизация работы с глобальной памятью

Поскольку глобальная память обладает высокой латентностью, важно рассмотреть способы доступа к ней и соответствующую оптимизацию доступа. Существуют 2 способа оптимизации работы с глобальной памятью:

- *выравнивание размеров* используемых типов;
- использование *объединенных запросов.*

3.1.1 Выравнивание размеров типов

Обращение к глобальной памяти происходит через чтение / запись 32/64/128-битовых слов. Крайне важным является то, что адрес, по которому происходит доступ, должен быть выровнен по размеру слова, то есть кратен размеру слова в байтах.

Так, если происходит чтение 32-битового слова по адресу 0, то потребуется одно обращение к памяти. Если чтение будет происходить с адреса 1, то потребуются 2 обращения к памяти, каждое из которых бу-

дет выровнено (первое читает по адресу 0, второе – по адресу 4), как показано на рисунке 16.

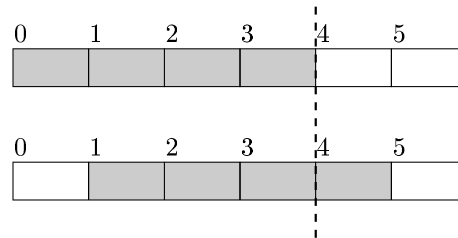

Рис. 16. Пример выровненного (сверху) и невыровненного (внизу) 4-байтового блока

Все функции, выделяющие глобальную память, выделяют ее выровненной по 256 байт. Однако проблемы с выравниванием могут возникнуть и в этом случае. Рассмотрим пример с использованием типа int3.

```
__device__ int3 data[512];
__global__ void initData(){
  int idx = threadIdx.x;
  data[idx] = make_int3(idx, idx, idx);
};
```

Несмотря на то что каждый элемент (длиной в 12 байт) помещается в 16 байтах, доступ к памяти не будет оптимальным. Хотя адрес первого элемента и выровнен по 16 байтам, адрес следующего элемента уже не выровнен, и его чтение потребует двух обращений. Лучше использовать тип int4 (16 байт), даже если 4-й компонент не нужен:

```
__device__ int4 data[512];
__global__ void initData(){
  int idx = threadIdx.x;
```

```
  data[idx] = make_int4(idx, idx, idx, 0);
};
```

В случае работы со структурами необходимо использовать спецификатор `__align__`, который позволяет выравнивать тип по заданному размеру:

```
struct __align__(16) vector3{
  float x;
  float y;
  float z;
};
```

Теперь все элементы массива будут находиться на адресах, кратных 16, что обеспечит чтение одного элемента за один раз. При таком способе оптимизации увеличился объем выделяемой и используемой памяти, но работа с этой памятью будет происходить быстрее.

3.1.2 Использование объединенных запросов

Важным для оптимизации работы с глобальной памятью является использование возможности GPU объединять несколько запросов к глобальной памяти в один (coalescing).

Все обращения мультипроцессора к памяти, в зависимости от Compute Capability, происходят независимо для каждой половины warp'а или для целого warp'а (Рис. 17). Максимальное объединение – это когда все запросы одного полу-warp'а или целого warp'а удается объединить в один большой запрос на чтение непрерывного блока памяти. Для того, чтобы это произошло, необходимо выполнение ряда условий. Сами условия за-

висят от версии архитектуры используемого GPU.

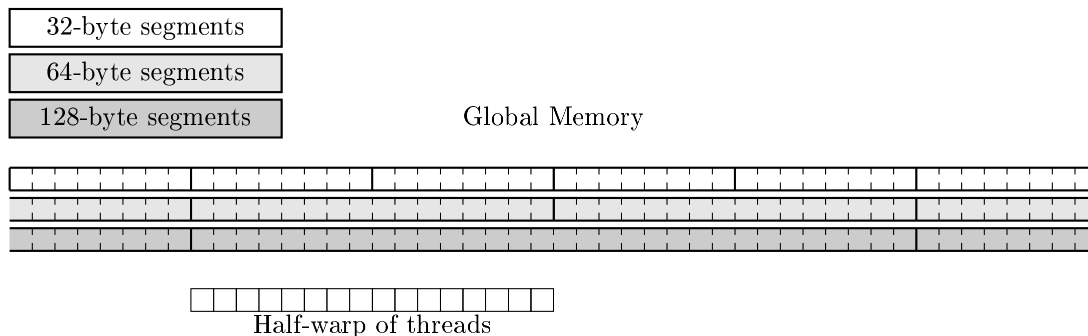

Рис. 17. Пример размещения данных в памяти. Тип float (32-bit)

Объединение для GPU с CC 1.0 и 1.1

Чтобы GPU с CC 1.0 или 1.1 произвел объединение запросов нитей половины warp'а, необходимо выполнение следующих условий:

- все нити обращаются к 32-битовым словам, образуя в результате один 64-байтовый блок, или все нити обращаются к 64-битовым словам, образуя в результате один 128-байтовый блок;
- получившийся блок выровнен по своем размеру, то есть адрес получившегося 64-байтового блока должен быть кратен 64, а адрес получившегося 128-байтового блока должен быть кратен 128;
- все 16 слов, к которым обращаются нити, лежат в пределах этого блока;
- нити обращаются к словам последовательно – k-ая нить должна обращаться к k-му элементу в блоке (при этом допускается, что отдельные нити пропустят обращение к соответствующим словам).

Если нити полу-warp'а не удовлетворяют какому-либо из данных условий, то каждое обращение к памяти происходит как отдельная транзакция. На рисунке 18 приведен типичный шаблон обращения к памяти, приводящий к объединению запросов в одну транзакцию. Для части нитей пропущено обращение к соответствующим словам.

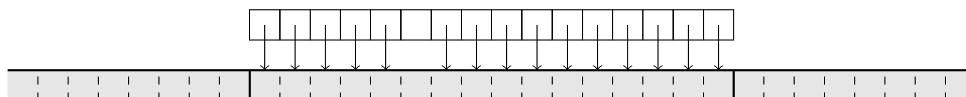

Рис. 18. Объединение – 1 транзакция

На рисунке 19 нарушен порядок обращения к словам, что приводит к чтению за 16 транзакций.

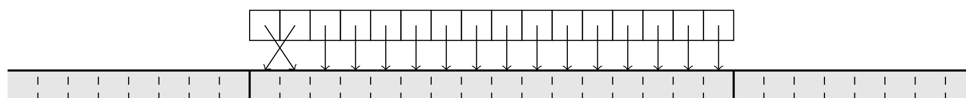

Рис. 19. Нарушена последовательность – 16 транзакций

На рисунке 20 нарушено условие выравнивания – хотя слова, к которым идет обращение, и образуют непрерывный блок из 64 байт, но начало этого блока не кратно его размеру. В этом случае чтение будет происходить также за 16 транзакций.

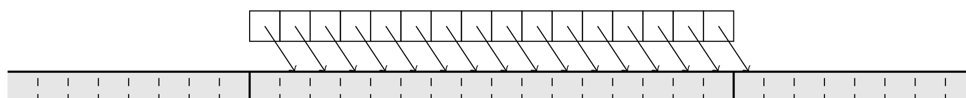

Рис. 20. Невыровненный адрес – 16 транзакций

Объединение для GPU с CC 1.2 и выше

Для GPU с CC 1.2 и выше объединение запросов в один будет происходить, если слова, к которым идет обращение нитей, лежат в одном сегменте размера 32 байта (если все нити обращаются к 8-битовым словам), 64 байта (если все нити обращаются к 16-битовым словам) и 128 байт (если все нити обращаются к 32-битовым или 64-битовым словам). Получающийся сегмент (блок) должен быть выровнен по 32/64/128 байтам. Порядок, в котором нити обращаются к словам, не имеет значения.

На рисунке 21 приведен шаблон доступа к глобальной памяти, приводящий к объединению запросов в одну транзакцию. Слова, к которым идет обращение нитей, лежат в одном сегменте 64 байт, а порядок обращения нитей к словам не имеет значения.

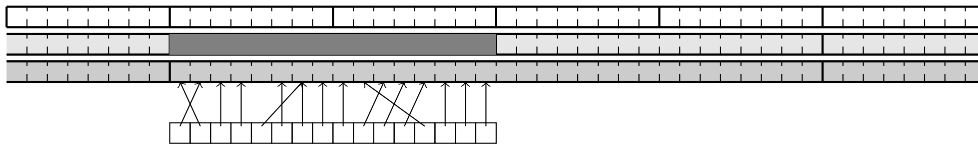

Рис. 21. 1 транзакция – 64 B сегмент

Запрос на рисунке 22 приведет к двум транзакциям. Слова, к которым идет обращение, лежат в разных сегментах: 64 байта и 32 байта.

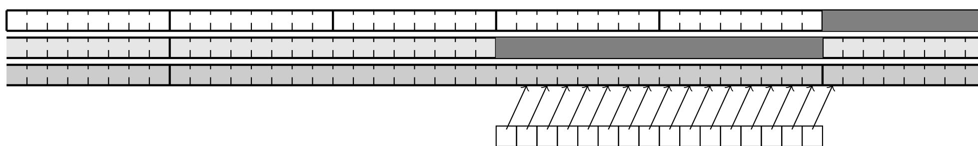

Рис. 22. 2 транзакции – 64 B и 32 B сегменты

Запрос на рисунке 23 произойдет за одну транзакцию. Слова, к которым идет обращение, лежат в одном сегменте 128 байт.

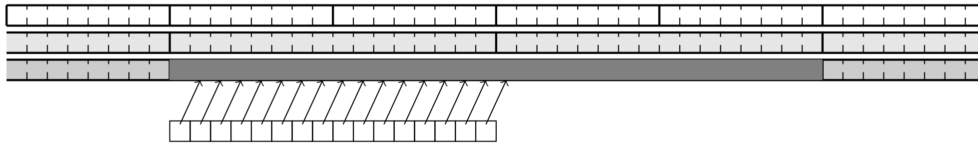

Рис. 23. 1 транзакция – 128 B сегмент

Объединение для GPU с CC 2.0 (Fermi)

Транзакции обрабатываются для warp'a (32 нити). Существуют 2 способа организации доступа к глобальной памяти, в зависимости от опции компилятора для включения/выключения L1 кэша.

- L1 кэш включен (по умолчанию опция `-Xptxas -dlcm=ca`):
  Транзакции выпускаются по 128 B. Они кэшируются в 16 kB или 48 kB L1 кэш на мультипроцессор. На рисунке 24 представлен пример обращения к глобальной памяти, приводящий к двум транзакциям по 128 байт.

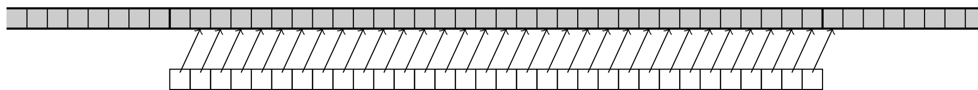

Рис. 24. 2 транзакции – 2*128 B сегмента

- L1 кэш выключен (опция `-Xptxas -dlcm=cg`):
  Транзакции выпускаются по 32 B. Такой способ обращения подходит, например, для разбросанных данных (Рис. 25).

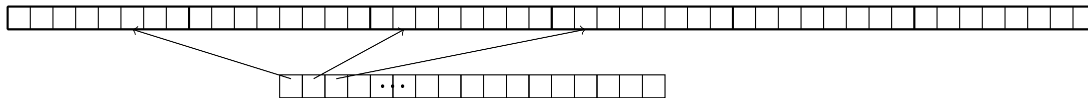

Рис. 25. 32 транзакции – 32*32 B сегмента, вместо 32*128 B сегмента

Объединение. Структуры

Эффективнее с точки зрения объединения запросов к памяти, использовать структуру массивов, а не массив структур.

```
struct A __align__(16){
    float a;
    float b;
    float c;
};

A array [1024];
...
A a = array [threadIdx.x];
```

В приведенном выше примере чтение структуры каждой нитью не даст объединения обращений к памяти, и на доступ к каждому элементу массива A понадобится отдельная транзакция. Однако, если вместо одного массива A можно сделать три массива его компонент, то ситуация изменится.

```
float a [1024];
float b [1024];
float c [1024];
...
float fa = a [threadIdx.x];
```

```
float fb = b [threadIdx.x];
float fc = c [threadIdx.x];
```

Каждый из трех запросов к очередной компоненте исходной структуры приведет к объединению запросов всех запросов нитей полу-warp'а, потребуется всего по 3 транзакции на полу-warp (вместо 16 транзакций ранее).

3.2 Оптимизация работы с разделяемой памятью

Как и при работе с глобальной памятью, при работе с разделяемой памятью есть свои шаблоны оптимального доступа к ней, обеспечивающие наибольшую скорость доступа.

3.2.1 Конфликты банков

Для повышения пропускной способности вся разделяемая память разбита на 16 *банков*, каждый из которых способен выполнить одно чтение или запись 32-битового слова.

Если в один банк придет сразу несколько обращений, то он должен будет выполнить их последовательно. Такая ситуация называется *конфликтом банков* и характеризуется *порядком конфликта* – максимальным числом обращений в банк. Если возникает конфликт второго порядка, то скорость доступа к разделяемой памяти снижается вдвое.

Поскольку обращение к разделяемой памяти происходит отдельно для каждого полу-warp'а, то необходимо отслеживать лишь конфликты банков в пределах каждого полу-warp'а.

Разбиение всей разделяемой памяти по банкам организовано следующим образом: подряд идущие 32-битовые слова попадают в подряд идущие банки. Если все 16 нитей полу-warp'a обращаются к 16 подряд идущим 32-битовым словам, то конфликта банков не возникает (Рис. 26).

0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64

bank 0	bank 1	bank 2	bank 3	bank 4	bank 5	bank 6	bank 7	bank 8	bank 9	bank 10	bank 11	bank 12	bank 13	bank 14	bank 15

Рис. 26. Разбиение разделяемой памяти на банки

На рисунке 27 приведены типичные шаблоны бесконфликтного доступа к разделяемой памяти.

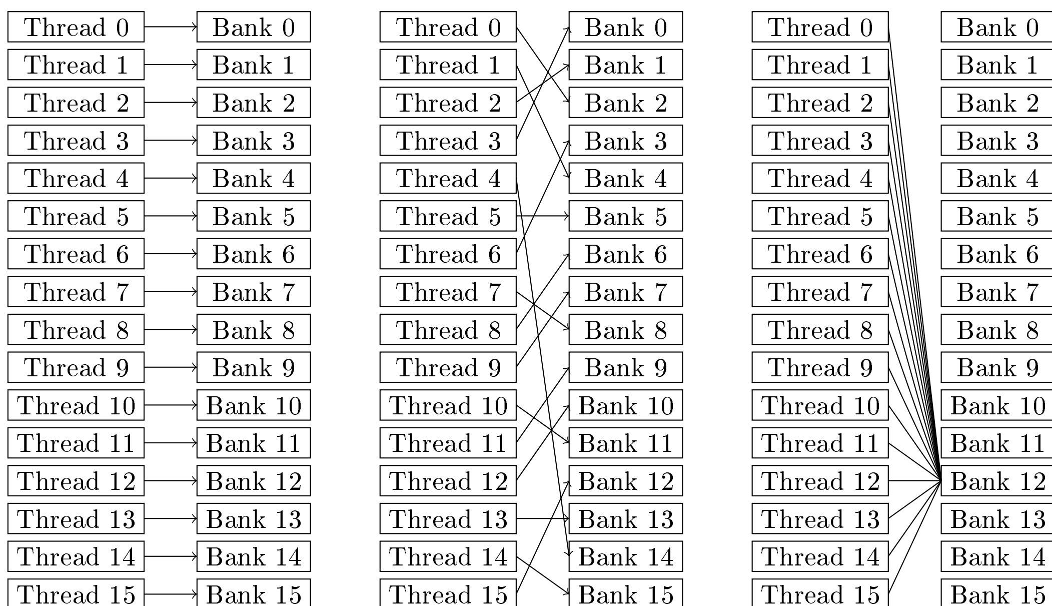

Рис. 27. Бесконфликтный доступ к разделяемой памяти

Возможен еще один вариант бесконфликтного доступа к разделяемой памяти – это когда все 16 нитей обращаются к одному и тому же адресу (broadcast) (Рис. 27, справа).

На рисунке 28 приведены шаблоны доступа к разделяемой памяти, приводящие к возникновению конфликтов по банкам памяти. Шаблон доступа слева приводит к появлению конфликтов 2-го порядка, шаблон доступа справа – к появлению конфликтов 4, 6 и 5-го порядков. Соответственно, для шаблона доступа слева скорость работы с разделяемой памятью снизится вдвое, а для шаблона справа – в 6 раз.

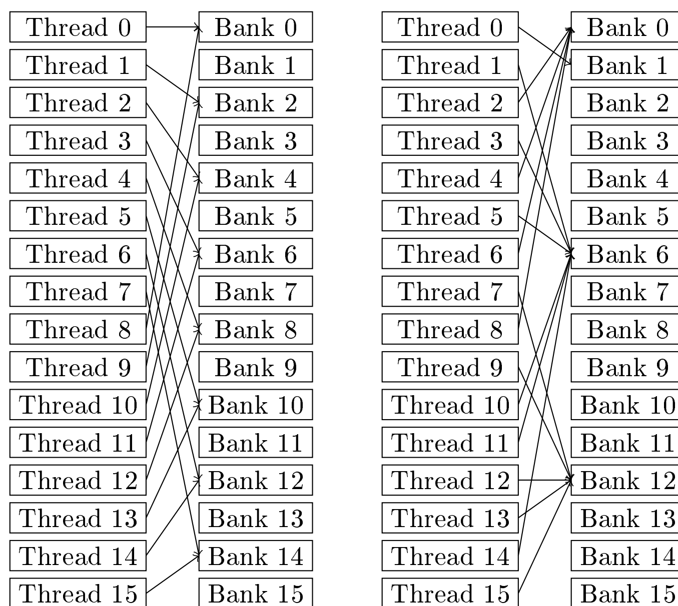

Рис. 28. Доступ к разделяемой памяти, в котором возникает конфликт

Ниже приведен простейший вариант бесконфликтного доступа к разделяемой памяти.

```
__shared__  float a[N];
float x = a [base + threadIdx.x];
```

Далее показаны случаи, когда адреса, по которым производится доступ в разделяемую память, линейно зависят от номера нити.

Конфликт 2-го порядка:

```
__shared__  short a[N];
short x = a [base + threadIdx.x];
```

Конфликт 4-го порядка:

```
__shared__  char a[N];
char x = a [base + threadIdx.x];
```

Конфликты 2-го порядка для устройств с cc = 1.x, нет конфликтов для cc = 2.x:

```
__shared__  double a[N];
double x = a [base + threadIdx.x];
```

В Fermi память разбивается на 32 банка, конфликты возникают на уровне warp'а.

Чтение и запись вещественных чисел двойной точности double на Fermi производятся по 64 бита, поэтому при последовательном доступе к массиву из double конфликты не возникают.

Конфликт банков возникает, если 2 или более нитей одного полуwarp'а обращаются к разным словам в одном банке.

3.2.2 Настройка размера разделяемой памяти (Fermi)

Fermi имеет 64KB памяти на мультипроцессор, приходящейся на L1 кэш и разделяемую память. Эта память может быть настроена следующим образом:

- 48 KB на L1 кэш и 16 KB на разделяемую память;
- 16 KB на L1 кэш и 48 KB на разделяемую память.

Функция `cudaDeviceSetCacheConfig()` определяет размер разделяемой памяти и L1 кэша.

```
cudaError_t cudaDeviceSetCacheConfig(
                    enum cudaFuncCache cacheConfig)
```

Параметр `cacheConfig` может принимать значения:

- `cudaFuncCachePreferNone` (автоматический выбор, по умолчанию);
- `cudaFuncCachePreferShared` (48 KB разделяемой памяти);
- `cudaFuncCachePreferL1` (48 KB L1 кэш).

Вопросы и задания

1. Для чего нужно выравнивание типов?
2. Для чего используется спецификатор `__align__` ?
3. Как происходит объединение запросов к глобальной памяти?
4. Опишите условия объединения запросов к глобальной памяти для устройств с cc=1.1 и cc=1.2.
5. Как организован доступ к глобальной памяти для устройств с cc=2.0?

6. Как L1 кэш влияет на доступ к глобальной памяти?
7. Какой способ хранения данных эффективней с точки зрения доступа к памяти: массив структур или структура массивов?
8. Что такое конфликт банков?
9. Опишите условия возникновения конфликта банков.
10. Приведите пример ядра, в котором возникает конфликт банков 4-го порядка.
11. С помощью какой команды настраивается размер разделяемой памяти в Fermi?

4 Реализация базовых операций

4.1 Параллельная редукция

Одной из часто встречающихся задач в алгоритмах является так называемая *редукция массива*. Пусть имеется некоторая бинарная ассоциативная операция $\oplus$ и массив $s = [a_0, a_1, .., a_{n-1}]$ из n элементов. Следующее выражение будет называться редукцией массива $s = [a_0, a_1, .., a_{n-1}]$ относительно заданной операции:

$$reduce(\oplus, s) = a_0 \oplus a_1 \oplus ... \oplus a_{n-1}.$$

Например,

$$reduce(+, [1\ 3\ 8\ 6\ 5\ 2\ 4\ 7]) = 36.$$

В качестве бинарной могут выступать такие операции, как сложение, умножение, минимум, максимум и другие.

Для распараллеливания операции редукции разобьем весь массив на пары и параллельно сложим элементы каждой пары между собой. В результате получим массив вдвое меньшей размерности, для которого повторим подобный процесс. После каждого повторения число элементов будет уменьшаться вдвое (Рис. 29).

Для реализации редукции на CUDA разобьем весь массив на части. Каждой части массива ставим в соответствие блок сетки (разбиваем массив поровну между всеми блоками). Задача блока – найти сумму всех элементов своей части и записать результирующее значение в выходной массив.

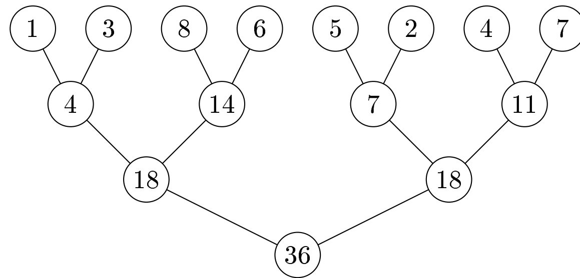

Рис. 29. Схема параллельного суммирования элементов массива

Стоит заметить, что в CUDA есть ограничение на количество блоков по каждому измерению. Размер сетки по каждому измерению не может превышать 65535. Использование одномерной сетки, что является наиболее подходящим способом организации блоков, накладывает ограничения на размер входного массива. Для больших массивов может понадобиться использование двумерной сетки.

Лимитирующим быстродействие фактором будет являться доступ к памяти, а не арифметические операции. Поэтому будем оптимизировать именно доступ к памяти. Каждый блок будет загружать свои элементы в разделяемую память и иерархически суммировать элементы уже в разделяемой памяти. При этом каждой нити соответствует один элемент массива. На рисунке 30 показано соответствие элементов массива (Values), нитей (ThreadI IDs) и шагов (St) для данного ядра.

Ниже приводится ядро, реализующее такой подход.

```
__global__ void reduce1(int * inData, int * outData){
    __shared__ int Data[BLOCK_SIZE];
    int tid = threadIdx.x;
    int i   = blockIdx.x * blockDim.x + threadIdx.x;
    // загрузить данные в разделяемую память
```

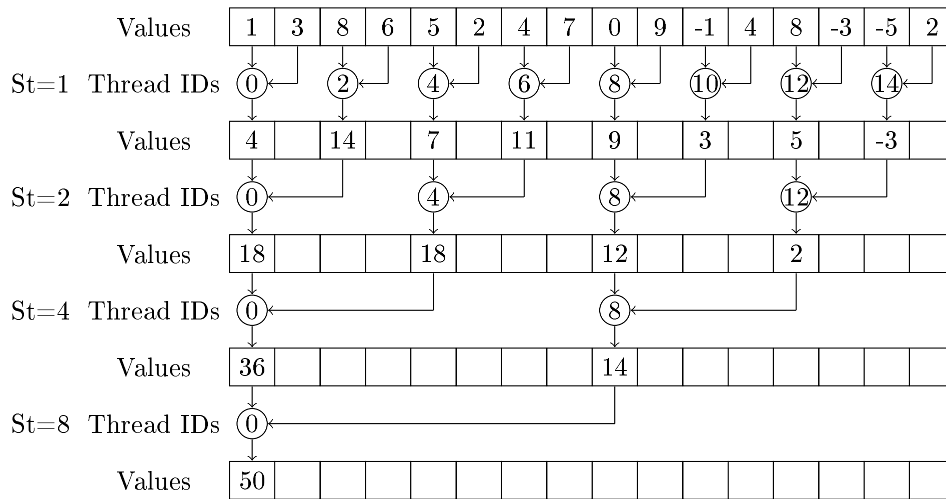

Рис. 30. Параллельная редукция. Вариант 1

```
    Data[tid] = inData[i];
    __syncthreads();
    for (int st = 1; st < blockDim.x; st*=2){
        if (tid % (2*st) == 0) // ветвление
            Data[tid] += Data[tid + st];
        __syncthreads();
    }
    if (tid == 0)
    // записать результат в разделяемую память
        outData[blockIdx.x] = Data[0];
}
```

При таком подходе условный оператор внутри цикла по `st` приводит к сильному ветвлению кода для каждого warp'а. Его можно избежать, перераспределив данные и операции по нитям (Рис. 31).

Ниже приводится соответствующее ядро.

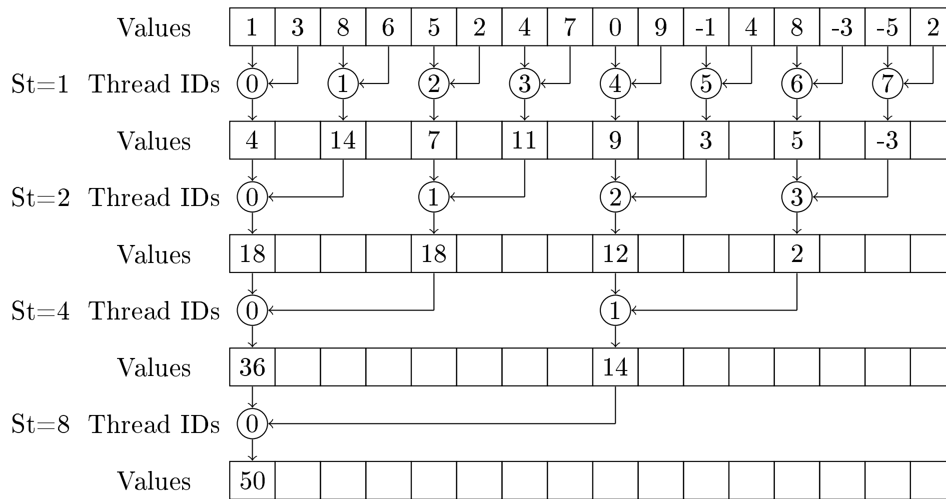

Рис. 31. Параллельная редукция. Вариант 2

```
__global__ void reduce2(int * inData, int * outData){
    __shared__ int Data[BLOCK_SIZE];
    int tid = threadIdx.x;
    int i   = blockIdx.x * blockDim.x + threadIdx.x;
    // загрузить данные в разделяемую память
    Data[tid] = inData[i];
    __syncthreads();
    for (int st = 1; st < blockDim.x; st*=2){
        int index = 2*st*tid;
        if (tid < blockDim.x)
            Data[index] += Data[index + st];
        __syncthreads();
    }
    if (tid == 0)
    // записать результат в глобальную память
        outData[blockIdx.x] = Data[0];
}
```

Для ядра `reduce2` на каждом шаге цикла по `st` будет не более одного ветвления, причем этом ветвление будет иметь место в последних итерациях цикла. Однако данная реализация приводит в большому числу конфликтов банков при обращении к разделяемой памяти. При `st=2` все обращения к разделяемой памяти придутся на банки с четными номерами (конфликт банков 2-го порядка). Аналогично при `st=4` все обращения придутся на банки с номерами 0, 4, 8 и 12 (конфликт банков 4-го порядка). Для каждого следующего шага цикла степень конфликта будет удваиваться.

Чтобы избавиться от этого, достаточно изменить порядок суммирования. Раньше суммирование начиналось с соседних элементов и расстояние увеличивалось вдвое. Теперь начнем суммирование с элементов, находящихся на расстоянии `BLOCK_SIZE/2`, и будем уменьшать расстояние между элементами вдвое (Рис. 32).

Ниже приводится соответствующий вариант ядра.

```
__global__ void reduce3(int * inData, int * outData){
    __shared__ int Data[BLOCK_SIZE];
    int tid = threadIdx.x;
    int i   = blockIdx.x * blockDim.x + threadIdx.x;
    // загрузить данные в разделяемую память
    Data[tid] = inData[i];
    __syncthreads();
    for (int st = blockDim.x / 2; st > 0; st>>=1){
        if (tid < st)
            Data[tid] += Data[tid + st];
        __syncthreads();
    }
    if (tid == 0)
```

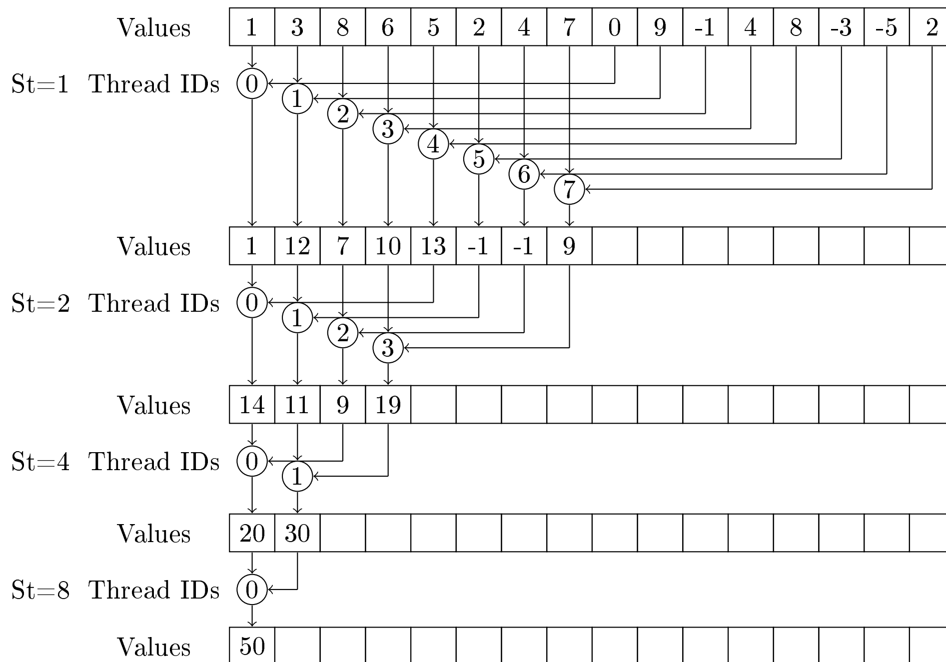

Рис. 32. Параллельная редукция. Вариант 3

```
    // записать результат в глобальную память
        outData[blockIdx.x] = Data[0];
}
```

В последней реализации ядра мы избавились от конфликтов по банкам, избавились от ветвления. Однако на первой итерации половина нитей простаивает. Для дальнейшей оптимизации уменьшим вдвое количество блоков, но при этом каждый блок будет обрабатывать вдвое больше элементов. Чтобы избежать увеличения требуемого объема разделяемой памяти, суммирование первых элементов будем производить сразу же при загрузке данных в разделяемую память.

Соответствующее ядро приводится ниже.

```
__global__ void reduce4(int * inData, int * outData){
    __shared__ int Data[BLOCK_SIZE];
    int tid = threadIdx.x;
    int i   = 2 * blockIdx.x * blockDim.x + threadIdx.x;
    // загрузить данные в разделяемую память
    Data[tid] = inData[i] + inData [i+blockDim.x];
    __syncthreads();
    for (int st = blockDim.x / 2; st > 0; st>>=1){
        if (tid < st)
            Data[tid] += Data[tid + st];
        __syncthreads();
    }
    if (tid == 0)
    // записать результат в глобальную память
        outData[blockIdx.x] = Data[0];
}
```

С увеличением `st` число активных нитей уменьшается. При `st`≤32 в каждом блоке останется по одному warp'у. Поэтому станут ненужными синхронизация и проверка `if tid < st`. Развернем цикл для `st`≤32. В результате получим следующую реализацию параллельной редукции массива.

```
__global__ void reduce5(int * inData, int * outData){
  __shared__ int Data[BLOCK_SIZE];
    int tid = threadIdx.x;
    int i = 2 * blockIdx.x * blockDim.x + threadIdx.x;
    // загрузить данные в разделяемую память
    Data[tid] = inData[i] + inData [i+blockDim.x];
    __syncthreads();
    for (int st = blockDim.x / 2; st > 32; s>>=1){
```

```
        if (tid < st)
            Data[tid] += Data[tid + st];
    __syncthreads();
    }
    if ( tid < 32 ){ // развернуть последние итерации
        Data [tid] += Data [tid + 32];
        Data [tid] += Data [tid + 16];
        Data [tid] += Data [tid + 8];
        Data [tid] += Data [tid + 4];
        Data [tid] += Data [tid + 2];
        Data [tid] += Data [tid + 1];
    }
    if (tid == 0)
    // записать результат в глобальную память
        outData[blockIdx.x] = Data[0];
}
```

Рассмотренное ядро строит отдельно суммы для каждой части массива, соответствующей отдельному блоку. В результате строится массив частичных сумм. Для подсчета суммы элементов массива исходный массив разбиваем на блоки, например, по 512 элементов. Используя ядро `reduce5`, параллельно находим суммы для каждого блока. В результате выполнения ядра получаем массив, где для каждого блока будет храниться сумма его элементов. Если в полученном массиве мало элементов, суммирование можно провести на CPU, иначе можно применить к нему операцию редукции и получить новый массив частичных сумм.

Ниже приводится код функции, осуществляющей суммирование элементов массива произвольного размера.

```
int reduce (int * Data, int n){
  int * s = NULL;
```

```
  int nBlocks = n/512;
  int res = 0;
  // выделить память под массив частичных сумм
  cudaMalloc((void **) &s, nBlocks * sizeof(int));
  // провести поблочную редукцию
  reduce5<<<dim3(nBlocks),dim3(BLOCK_SIZE)>>>(Data,s);
  if (nBlocks > BLOCK_SIZE)
    res = reduce(s,nBlocks);
  else {
    int * h_s = new int [nBlocks];
    cudaMemcpy(h_s,s,nBlocks * sizeof(int), cudaMemcpyDeviceToHost);
    for (int i = 0; i < nBlocks; i++)
      res += h_s[i];
    delete [] h_s;
  }
  cudaFree(s);
  return res;
}
```

Если известно число итераций на этапе компиляции, можно полностью развернуть цикл последней итерации редукции. Размер блока ограничен GPU до 1024 нитей. Будем рассматривать блоки, размеры которых равны степеням двойки. Тогда можно развернуть цикл для фиксированного размера блока. Однако размер блока не известен на этапе компиляции. Решение проблемы заключается в использовании шаблонов функций.

CUDA поддерживает C++ параметры шаблона на `device` и `host` функциях. Установим размер блока как параметр шаблона функции.

```
template <int BLOCK_SIZE>
__global__ void reduce6 (int * inData, int * outData){
```

```
...
  if (BLOCK_SIZE >= 512)
    {if (tid < 256)  {Data[tid]+=Data[tid+256];}
    __syncthreads();}
  if (BLOCK_SIZE >= 256)
    {if (tid < 128)  {Data[tid]+=Data[tid+128];}
    __syncthreads();}
  if (BLOCK_SIZE >= 128)
    {if (tid < 64)  {Data[tid]+=Data[tid+64];}
    __syncthreads();}
  if (tid < 32){
    if (BLOCK_SIZE >= 64) Data [tid] += Data [tid + 32];
    if (BLOCK_SIZE >= 32) Data [tid] += Data [tid + 16];
    if (BLOCK_SIZE >= 16) Data [tid] += Data [tid + 8];
    if (BLOCK_SIZE >= 8)  Data [tid] += Data [tid + 4];
    if (BLOCK_SIZE >= 4)  Data [tid] += Data [tid + 2];
    if (BLOCK_SIZE >= 2)  Data [tid] += Data [tid + 1];
  }
...
}
```

Опишем вызов ядра для всех размеров блока:

```
switch(threads)
{
  case 512:
  reduce6<512><<<dimGrid,dimBlock>>>(inData,outData);
  break;
  case 256:
  reduce6<256><<<dimGrid,dimBlock>>>(inData,outData);
  break;
  case 128:
  reduce6<128><<<dimGrid,dimBlock>>>(inData,outData);
  break;
  case 64:
```

```
  reduce6< 64><<<dimGrid,dimBlock>>>(inData,outData);
  break;
  case 32:
  reduce6< 32><<<dimGrid,dimBlock>>>(inData,outData);
  break;
  case 8:
  reduce6<  8><<<dimGrid,dimBlock>>>(inData,outData);
  break;
  case 4:
  reduce6<  4><<<dimGrid,dimBlock>>>(inData,outData);
  break;
  case 2:
  reduce6<  2><<<dimGrid,dimBlock>>>(inData,outData);
  break;
  case 1:
  reduce6<  1><<<dimGrid,dimBlock>>>(inData,outData);
  break;
}
```

Можно рассмотреть последний вариант оптимизации. Заменим сложение и загрузку двух элементов

```
int tid = threadIdx.x;
int i = 2 * blockIdx.x * blockDim.x + threadIdx.x;
Data[tid] = inData[i] + inData [i+blockDim.x];
__syncthreads();
```

на цикл `while` для добавления необходимого числа элементов:

```
int tid = threadIdx.x;
int i = 2 * blockIdx.x * BLOCK_SIZE + threadIdx.x;
int gridSize = BLOCK_SIZE * 2 * gridDim.x;
Data[tid] = 0;
```

```
while (i<n){
    Data[tid] = inData[i] + inData [i+BLOCK_SIZE];
    i += gridSize;
}
__syncthreads();
```

Ниже приводится последний вариант ядра.

```
template <int BLOCK_SIZE>
__global__ void reduce7(int * inData, int *outData, int n)
{
  extern __shared__ int Data[];
  int tid = threadIdx.x;
  int i = blockIdx.x*(BLOCK_SIZE*2) + tid;
  int gridSize = BLOCK_SIZE*2*gridDim.x;
  Data[tid] = 0;
  do{
    Data[tid] += inData[i] + inData[i+BLOCK_SIZE];
    i += gridSize;
  }
  while (i < n);
  __syncthreads();
  if (BLOCK_SIZE >= 512)
    {if (tid < 256) {Data[tid] += Data[tid + 256];}
    __syncthreads();}
  if (BLOCK_SIZE >= 256)
    {if (tid < 128) { Data[tid] += Data[tid + 128];}
    __syncthreads();}
  if (BLOCK_SIZE >= 128)
    {if (tid < 64)  {Data[tid] += Data[tid + 64];}
    __syncthreads();}
  }
  if (tid < 32){
    if (BLOCK_SIZE >= 64) Data[tid] += Data[tid + 32];
    if (BLOCK_SIZE >= 32) Data[tid] += Data[tid + 16];
```

```
    if (BLOCK_SIZE >= 16) Data[tid] += Data[tid + 8];
    if (BLOCK_SIZE >= 8) Data[tid] += Data[tid + 4];
    if (BLOCK_SIZE >= 4) Data[tid] += Data[tid + 2];
    if (BLOCK_SIZE >= 2) Data[tid] += Data[tid + 1];
 }
  if (tid == 0) outData[blockIdx.x] = Data[0];
}
```

4.2 Транспонирование матрицы

Рассмотрим задачу транспонирования матрицы размером $w \times h$. Ниже приводится соответствующий код, реализованный на CPU.

```
__host__ void transposeMatrixCPU(float * inputMatrix,
                    float * outputMatrix, int w, int h){
  for (int y = 0; y < h; y++){
    for (int x = 0; x < w; x++){
      outputMatrix[x * h + y] = inputMatrix[y * w + x];
  }}
}
```

Разобьем исходную матрицу на двумерные блоки. Каждый блок будет обрабатывать свою подматрицу (Рис. 33). Ниже приведен код, реализующий простейший вариант транспонирования матрицы на GPU.

```
__global__ void transposeMatrixGPU(float * inputMatrix,
                    float * outputMatrix, int w, int h){
  int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
  int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
```

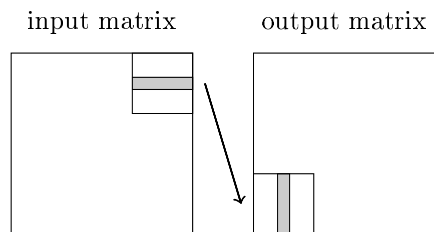

Рис. 33. Транспонирование матрицы. Глобальная память

```
  if ((xIndex < w) && (yIndex < h)){
    int inputIdx = xIndex + w * yIndex;
    int outputIdx = yIndex + h * xIndex;
    outputMatrix[outputIdx] = inputMatrix[inputIdx];
  }
}
```

Чтение в глобальную память будет происходить с объединением запросов, а запись – нет (доступ осуществляется по столбцам).

Загрузим подматрицу в разделяемую память. Блок нитей считывает подматрицу, записывает ее в разделяемую память, вычисляет новые индексы, записывает результат транспонирования. (Рис. 34).

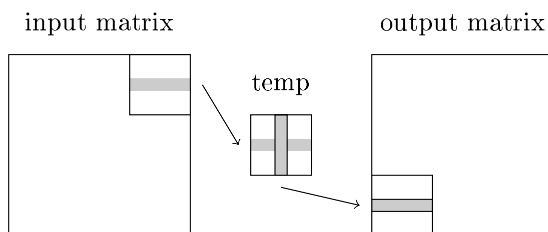

Рис. 34. Транспонирование матрицы. Разделяемая память

Ниже приводится соответствующий вариант транспонирования матрицы с использованием разделяемой памяти.

```
#define BLOCK_SIZE 16
__global__ void transposeMatrixFast(float * inputMatrix,
                    float * outputMatrix, int w, int h){
  __shared__ float temp[BLOCK_SIZE][BLOCK_SIZE];

  int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
  int yIndex = blockIdx.y * blockDim.y + threadIdx.y;

  if ((xIndex < w) && (yIndex < h)){
    int idx = yIndex * w + xIndex;
    temp[threadIdx.y][threadIdx.x] = inputMatrix[idx];
  }
  __syncthreads();

  xIndex = blockIdx.y * blockDim.y + threadIdx.x;
  yIndex = blockIdx.x * blockDim.x + threadIdx.y;

  if ((xIndex < h) && (yIndex < w)){
    int idx = yIndex * h + xIndex;
    outputMatrix[idx] = temp[threadIdx.x][threadIdx.y];
  }
}
```

Для матрицы размера 16×16 все элементы столбца лежат в одном банке, что приводит к конфликту банков 16-го порядка. Решением этого конфликта будет добавление дополнительного столбца в разделяемую память:

```
__shared__ float temp[BLOCK_SIZE][BLOCK_SIZE+1];
```

Тогда все элементы строки (кроме последнего, не используемого), а также все элементы столбца будут лежать в разных банках. Данные, лежащие на одной антидиагонали, будут находиться в одном банке. Таким образом, за счет небольшого увеличения памяти удалось полностью избавиться от конфликтов по банкам памяти.

Если на устройствах с cc<2.0 используется сдвиг

```
__shared__ float temp[16][17];
```

для Fermi нужно поменять и размер массива и сдвиг с учетом размера warp'a:

```
__shared__  float temp[32][33];
```

4.3 Умножение матрицы на вектор

В качестве следующего примера рассмотрим задачу умножения матрицы A размера $w \times h$ на вектор x:

$$y = Ax.$$

Каждая нить вычисляет единственное выходное значение y – произведение одной строки A и вектора x. Вычисления полностью независимы (Рис. 35).

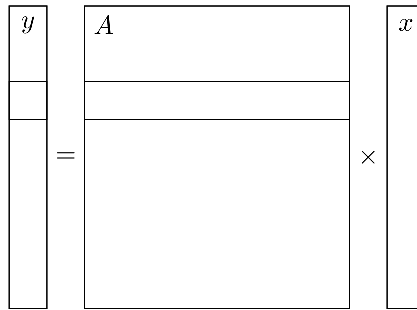

Рис. 35. Умножение матрицы на вектор. Глобальная память

Ниже приводится соответствующее ядро с использованием глобальной памяти, а также функция, вызывающая это ядро.

```
__global__ void MatVec1(float * y, const float* A,
                        const float* x, int w, int h){
  // вычисление индекса нити
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < h){
    float res = 0.;
    // умножение одной строки
    for (int j = 0; j < w; ++j){
      res += A[i*w + j] * x[j];
    }
    // записать результат в глобальную память
    y[i] = res;
  }
}
int main(){
  // выделить память
```

```
  float* d_A, d_x, d_y;
  cudaMalloc((void**)&d_A, w*h* sizeof(float));
  cudaMalloc((void**)&d_x, w* sizeof(float));
  cudaMalloc((void**)&d_y, h* sizeof(float));
  // загрузить A и x
  cudaMemcpy(d_A, h_A, w*h* sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_x, h_x, w* sizeof(float),   cudaMemcpyHostToDevice);
  // вычислить размерности блока и сетки
  dim3 threadBlock(MAX_THREADS, 1 );
  dim3 blockGrid(h / MAX_THREADS + 1, 1, 1);
  MatVec1<<< blockGrid, threadBlock >>>(d_y, d_A, d_x, w,h);
  cudaThreadSynchronize();
  // загрузить результат y
  cudaMemcpy(h_y, d_y, h* sizeof(float), cudaMemcpyDeviceToHost);
  cudaFree(d_A);
  cudaFree(d_x);
  cudaFree(d_y);
  return 0;
}
```

Эта программа не является оптимальной. Каждая нить загружает из глобальной памяти w элементов из A, w элементов из x. Все нити получают доступ к одному и тому же элементу из x. Поэтому можно загрузить x в разделяемую память (Рис. 36).

Ниже приводится код программы, реализующей умножение матрицы на вектор, с использованием разделяемой памяти.

```
__global__ void MatVec2(float * y, const float * A,
           const float * x, int w, int h, int nIter){
  extern __shared__ float vec[];
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float res = 0.; int vOffs = 0;
  // загрузить x в разделяемую память
```

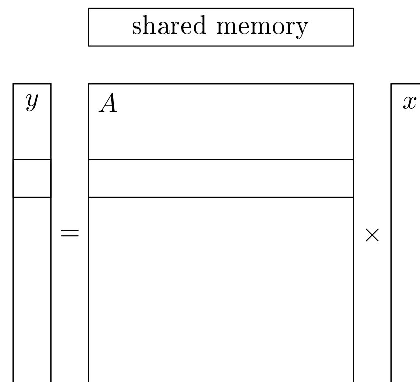

Рис. 36. Умножение матрицы на вектор. Разделяемая память

```
  for (int iter = 0; iter < nIter; ++iter, vOffs += blockDim.x){
    vec[vOffs + threadIdx.x] = x[vOffs + threadIdx.x];
  }
  __syncthreads();
  // вычислить произведения
  if (i<h){
    for (int j = 0; j < w; ++j){
      res += A[vOffs+j] * vec[j];
      y[i] = res;
    }
  }
}
int main(){
...
  dim3 threadBlock(MAX_THREADS, 1);
  dim3 blockGrid(h / MAX_THREADS + 1, 1, 1);
  MatVec2<<< blockGrid, threadBlock, w* sizeof(float)>>>
            (d_y, d_A, d_x, w,h, w / MAX_THREADS);
...
  return 0;
}
```

4.4 Умножение матриц

В следующем примере рассмотрим умножение квадратных матриц A и B размера $w \times w$:

$$C = A * B.$$

Простейший вариант использует по одной нити на каждый элемент получающейся матрицы C. При этом каждая нить загружает строку матрицы A, загружает столбец матрицы B и производит требуемые вычисления. Размер матриц при таком варианте ограничен максимальным числом нитей в блоке (Рис. 37).

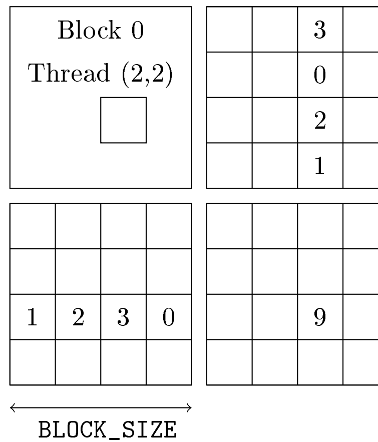

Рис. 37. Умножение квадратных матриц. Один блок

Для вычисления произведения матриц произвольного размера разобьем матрицы на блоки. Каждый блок вычисляет подматрицу размера `BLOCK_SIZE` $\times$ `BLOCK_SIZE` результирующей матрицы.

Сгенерируем 2D сетку из $(w/\text{BLOCK_SIZE})^2$ блоков (Рис. 38).

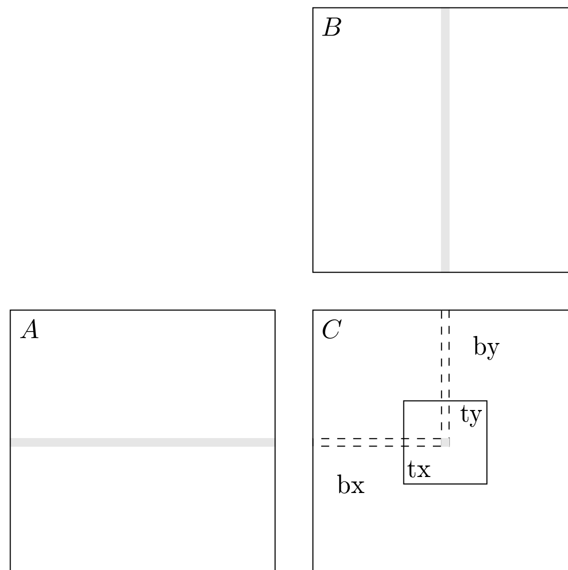

Рис. 38. Умножение квадратных матриц. Произвольный размер

Ниже приводится соответствующий код.

```
__global__ void matMult(float * C, float * A, float * B, int w)
{
  // индекс блока
   int bx = blockIdx.x;    int by = blockIdx.y;
  // индекс нити
  int tx = threadIdx.x;    int ty = threadIdx.y;
  // результат
  float sum = 0.;
  // индекс A[i][0]
  int ia = w * BLOCK_SIZE * by + w * ty;
  // индекс B[0][j]
  int ib = BLOCK_SIZE * bx + tx;
  for (int k = 0, k < w; k++)
  {
    sum += A[ia + k] * B[ib + k * w];
```

```
  }
  // индекс C[i][j]
  int ic = w * BLOCK_SIZE * by * BLOCK_SIZE *bx;
  // записать результат в глобальную память
  C[ic + w * ty + tx] = sum;
}
```

Элементы матриц A и B загружаются w раз из глобальной памяти. Для вычисления одного элемента произведения матриц нужно выполнить $2w$ арифметических операций и $2w$ чтений из глобальной памяти. В данном случае основным лимитирующим фактором является скорость доступа к глобальной памяти.

Можно повысить быстродействие программы за счет использования разделяемой памяти. Будем рассматривать матрицы произвольного размера при условии, что размерности A и B кратны `BLOCK_SIZE`. Загрузим подматрицы A и B в разделяемую память. Каждая нить читает один элемент подматрицы A и один элемент подматрицы B. Используем каждую подматрицу для всех элементов C. Выполним внешний цикл по подматрицам (Рис. 39).

Ниже приводится соответствующий код.

```
__global__ void matMult(float* C, float* A, float* B,
                        int wA, int wB)
{
  // индекс блока
  int bx = blockIdx.x;     int by = blockIdx.y;
  // индекс нити
  int tx = threadIdx.x;    int ty = threadIdx.y;
  // индекс первой подматрицы A
  int aBegin = wA * BLOCK_SIZE * by;
```

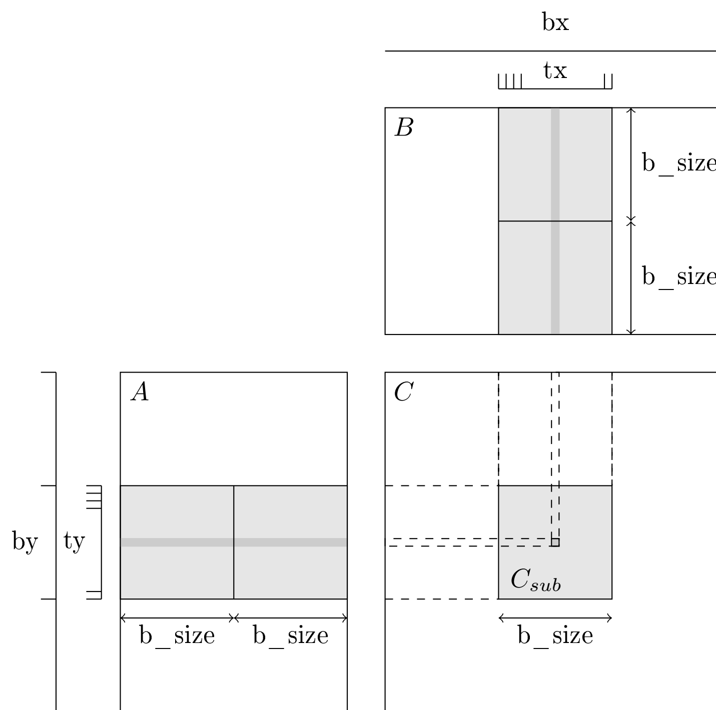

Рис. 39. Умножение матриц. Разделяемая память

```
// индекс последней подматрицы A
int aEnd = aBegin + wA - 1;
// шаг перебора подматриц A
int aStep = BLOCK_SIZE;
// индекс первой подматрицы B
int bBegin = BLOCK_SIZE * bx;
// шаг перебора подматриц B
int bStep = BLOCK_SIZE * wB;
// Csub используется для хранения вычисляемых элементов
float Csub = 0;
// цикл по всем под-матрицам A и B
for (int a = aBegin, b = bBegin; a <= aEnd;
     a += aStep, b += bStep)
{
```

```
    // объявление массивов As, Bs в разделяемой памяти
    // для хранения подматриц A, B
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    // загрузить матрицы в разделяемую память
    // каждая нить загружает по одному элементу каждой матрицы
    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];
    // убедимся, что все матрицы загружены
    __syncthreads();
    // умножим две матрицы;
    // каждая нить вычисляет один элемент подматрицы
    for (int k = 0; k < BLOCK_SIZE; ++k)
    {
      Csub += As[ty][k] * Bs[k][tx];
    }
    // убедимся, что предыдущие вычисления завершены
    // перед загрузкой двух новых подматриц A и B
    __syncthreads();
  }
  // записать результат;
  // каждая нить записывает один элемент
  int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
  C[c + wB * ty + tx] = Csub;
}
```

4.5 Решение трехдиагональных систем уравнений

Рассмотрим систему n линейных алгебраических уравнений с трехдиагональной матрицей:

$$Ax = \begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & 0 & \\ & a_3 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & \ddots & \ddots & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{n-1} \\ f_n \end{pmatrix} = f.$$

Идея алгоритма *циклической редукции* заключается в исключении переменных из соседних уравнений и рекурсивном сокращении системы до одного уравнения или системы из двух уравнений.

Рассмотрим i-е уравнение совместно с $i-1$ и $i+1$ уравнениями:

$$\begin{aligned} a_{i-1}x_{i-2} + b_{i-1}x_{i-1} + c_{i-1}x_i =& f_{i-1}, \\ a_i x_{i-1} + b_i x_i + c_i x_{i+1} =& f_i, \\ a_{i+1}x_i + b_{i+1}x_{i+1} + c_{i+1}x_{i+2} =& f_{i+1}. \end{aligned}$$

Чтобы исключить x_{i-1} и x_{i+1}, умножим первое уравнение на $\alpha_i = -a_i/b_{i-1}$, последнее уравнение – на $\gamma_i = -c_i/b_{i+1}$ и просуммируем три уравнения:

$$a_i^{(1)}x_{i-2} + b_i^{(1)}x_i + c_i^{(1)}x_{i+2} = f_i,$$

где коэффициенты вычисляются следующим образом:

$$a_i^{(1)} = \alpha_i a_{i-1}, \quad b_i^{(1)} = b_i + \alpha_i c_{i-1} + \gamma_i a_{i+1},$$
$$c_i^{(1)} = \gamma_i c_{i+1}, \quad f_i^{(1)} = f_i + \alpha_i f_{i-1} + \gamma_i f_{i+1}.$$

В уравнения с четными номерами входят неизвестные только с четными номерами. В полученной подсистеме остается $n/2$ уравнений.

Циклическая редукция решает трехдиагональную систему за два этапа: прямой и обратный ход. Пусть $n = 2^q - 1$, $q \in \mathbb{N}$. Прямой ход вычисляет новые коэффициенты и правые части для уровней редукции $l = 1, ..., q-1$:

$$\begin{aligned} a_i^{(l)} &= \alpha_i a_{i-2^{l-1}}^{(l-1)}, \\ b_i^{(l)} &= b_i^{(l-1)} + \alpha_i c_{i-2^{l-1}}^{(l-1)} + \beta_i a_{i+2^{l-1}}^{(l-1)}, \\ c_i^{(l)} &= \beta_i c_{i+2^{l-1}}^{(l-1)}, \\ f_i^{(l)} &= f_i^{(l-1)} + \alpha_i f_{i-2^{l-1}}^{(l-1)} + \beta_i f_{i+2^{l-1}}^{(l-1)}, \end{aligned} \tag{1}$$

где

$$\alpha_i = -a_i / b_{i-2^{l-1}}, \quad \beta_i = -c_i / b_{i+2^{l-1}},$$
$$i \in \{k2^l \,|\, k = 1, .., 2^{q-2l}\}.$$

Каждый шаг редукции уменьшает число уравнений вдвое. На уровне $q-1$ остается одно уравнение. Обратный ход рекурсивно находит решение и выполняется для уровней $l = q, q-1, ..., 1$:

$$x_i = \frac{1}{b_i^{(l-1)}} \left(f_i^{(l-1)} - a_i^{(l-1)} x_{i-2^{l-1}} - c_i^{(l-1)} x_{i+2^{l-1}} \right), \qquad (2)$$

$$i \in \{2^{l-1} + k2^l \mid k = 1, ..., 2^{q-l} - 1\}, x_0 = x_{2^q} = 0.$$

На рисунке 40 представлена схема циклической редукции.

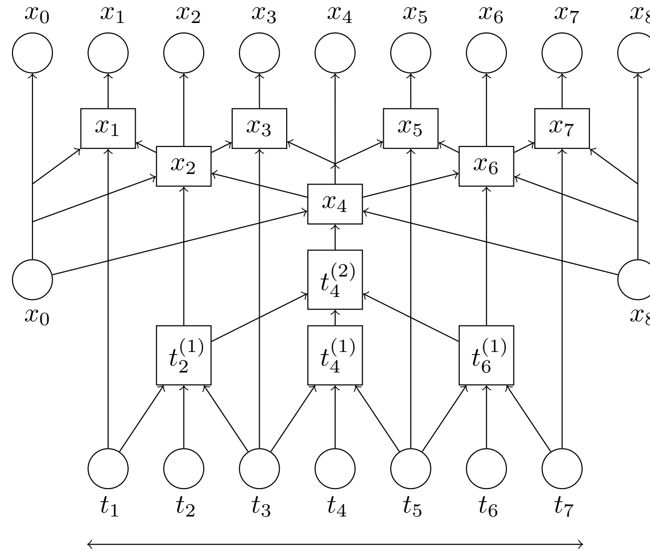

Рис. 40. Схема циклической редукции

Значения $t_i = (a_i, b_i, c_i, f_i)$ – коэффициенты i-й строки. Значения $t_0 = (0, 1, 0, 0)$ – граничные условия. Значения $t_i^{(l)} = (a_i^{(l)}, b_i^{(l)}, c_i^{(l)}, f_i^{(l)})$ – вычисленные коэффициенты i-й строки на l-м уровне редукции. Квадраты соответствуют вычислениям уравнений (1) на первом этапе редукции (прямой ход), прямоугольники – вычислениям уравнений (2) на втором этапе (обратный ход).

Количество параллельных потоков уменьшается при прямом ходе и увеличивается при обратном ходе с каждым уровнем l. Рассмотрим ва-

риацию алгоритма циклической редукции – параллельную циклическую редукцию (Рис. 41). Применим алгоритм циклической редукции ко всем n уравнениям. В результате алгоритм требует больше вычислений, однако он обладает более высоким параллелизмом. Преимуществом этого алгоритма является лучший доступ к памяти.

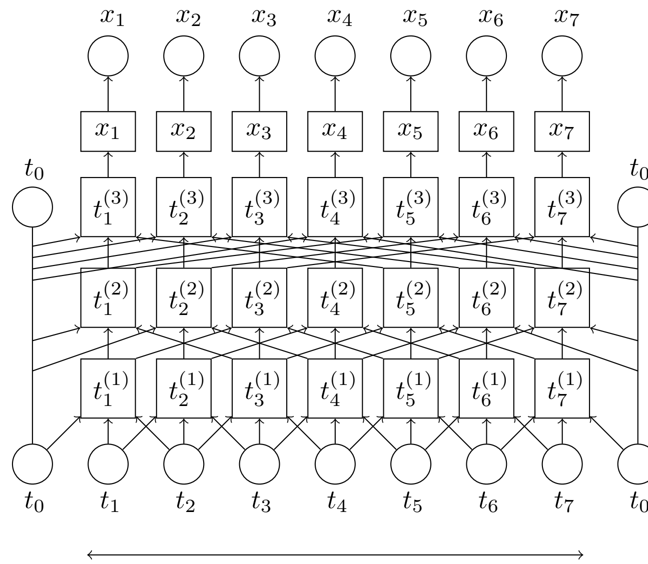

Рис. 41. Схема параллельной циклической редукции

Ниже приведен код, реализующий параллельную циклическую редукцию. Из-за наличия ограничений на максимальный размер блока в 1024 нитей, максимальный размер системы уравнений `dim`$\leq$ 1024.

Необходимо хранить в памяти три вектора `l`, `d`, `u` для системы (нижняя диагональ, диагональ, верхняя диагональ) и один вектор `f` для правой части, в котором также будет находиться вектор решения. Ядро использует `l`, `d`, `u` для хранения рекурсивно вычисляемых коэффициентов, поэтому дополнительного хранения временных переменных не тре-

буется. Все три вектора могут быть помещены в разделяемую память. В примере `tid` это номер нити в блоке.

```
 __device__ void pcr(int tid, int dim, float * l,
                   float * d, float * u, float * f){
  float lTemp, uTemp, fTemp;
  for (int i = 1; i < dim; i *= 2){
   lTemp=0; uTemp=0;
   if (tid < dim){
      if (tid - i >=0)
        if (d[tid-i] != 0)
            lTemp = -l[tid]/d[tid-i];
      if (tid + i < dim)
        if (d[tid+i] != 0)
            uTemp =  -u[tid]/d[tid+i];
      fTemp = f[tid];
    }
    __syncthreads();
 if (tid < dim){
  if (tid - i >= 0)
  {
   d[tid]  += lTemp * u[tid-i];
   fTemp   += lTemp * f[tid-i];
   lTemp   *= l[tid-i];
   }
  if (tid + i < dim)
  {
   d[tid]  += uTemp * l[tid+i];
   fTemp   += uTemp * f[tid+i];
   uTemp   *= u[tid+i];
  }
 }
 __syncthreads();
 if (tid < dim){
```

```
   l[tid] = lTemp;
   u[tid] = uTemp;
   f[tid] = fTemp;
  }
  __syncthreads();
 }
 if (tid < dim)
  f[tid] /= d[tid];
 __syncthreads();
}
```

Если размерность уравнений `dim > 1024`, одна нить должна обрабатывать несколько строк. Заменим секции кода

```
if (tid < dim) {...}
```

на цикл

```
for (int elem = tid; elem < dim; elem += size)
{...}
```

где `size` – количество нитей в блоке. Так как порядок выполнения нитей не гарантируется, необходимо создавать группу временных переменных `lTemp`, `uTemp`, `fTemp` на каждой итерации цикла. Необходимо также обратить внимание на размещение векторов `l`, `d`, `u`, `f` в памяти устройства. Векторы, которые не помещаются в разделяемую память, должны быть размещены в глобальной.

4.6 Конечно-разностные методы

Рассмотрим пример, часто встречающийся в конечно-разностных методах.

```
__global__ void diff(int * input, int * output){
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i>0){
    // сколько раз ядро загружает input[i]?
    int x_i = input[i];// один раз нитью i
    int x_i1 = input[i-1];// второй раз нитью i+1
    output[i] = x_i - x_i1;
  }
}
```

Для того чтобы избежать повторного чтения данных из глобальной памяти, загрузим их в разделяемую память.

```
__global__ void diff(int * input, int * output){
  __shared__ int s[BLOCK_SIZE];
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  int tid = threadIdx.x;
  s[tid] = input[i];
  __syncthreads();

  if (tid >0)
    output[i] = s[tid] - s[tid-1];
  else if (i>0) // граница блока
    output[i] = s[tid] - input[i-1];
}
```

Вопросы и задания

1. Приведите примеры алгоритмов, в которых используется операция редукции массива.
2. Постройте график времени выполнения программы параллельной редукции для 7 рассмотренных вариантов для различной размерности входных массивов. Посчитайте ускорение работы программы. Сравните время выполнения с CPU.
3. Напишите программу, реализующую скалярное произведение двух векторов, нормы вектора.
4. Напишите программу, реализующую поиск максимального элемента массива.
5. Напишите программу, реализующую умножение матрицы на ее транспонированную.

5 Оптимизация приложений

5.1 Математические функции

CUDA поддерживает все математические функции из стандартной библиотеки языка C. Большинство стандартных математических функций используют числа с двойной точностью (`double`). Вычисления с двойной точностью выполняются на GPU медленнее, чем вычисления с одинарной точностью. Если не требуется двойная точность, то можно использовать `float`-аналоги стандартных функций. `float`-аналогом функции `sin` является функция `sinf`. Кроме того, CUDA предоставляет специальный набор функций пониженной точности, но обеспечивающих высокое быстродействие. Таким аналогом для функции вычисления синуса является `__sinf`.

Итак, в CUDA существуют два типа математических операций одинарной точности:

- `funcf()`: медленная, но более точная реализация;
  Примеры: `sinf(x)`, `expf(x)`, `powf(x,y)`
- `__funcf()`: быстрая, но менее точная реализация.
  Примеры: `__sinf(x)`, `__expf(x)`, `__powf(x,y)`

Опция компилятора `-use_fast_math` заменяет все вызовы `funcf()` на `__funcf()`.

Вычисления с двойной точностью поддерживаются на устройствах с cc=1.3 и выше. Это означает, что при компиляции нужно указывать

версию архитектуры, например `-arch=compute_20.` В Makefile от SDK в секции `CUFILES` следует добавить признак архитектуры: `CUFILES_sm_20.`

В C/C++ числа без суффикса `f`, например, `1.45`, являются числами двойной точности. Используйте суффикс `f`, если вычисления должны проводиться с одинарной точностью: `1.45f.`

Деление целых чисел и нахождение остатка от деления являются дорогостоящими операциями, и должны быть заменены на побитовые операции. Если n – это степень двойки, тогда i/n эквивалентно $i >> log_2(n)$ и $i\,\%n$ эквивалентно $i\,\&(n-1)$.

5.2 Использование потоков

Многие функции CUDA runtime API – асинхронные, то есть управление CPU возвращается еще до реального завершения требуемой операции. К числу асинхронных операций относятся:

- запуск ядра;
- функции копирования памяти, имена которых оканчиваются на Async;
- функция копирования памяти DeviceToDevice;
- функция инициализации памяти.

Поток (CUDA stream) – это последовательность команд, выполняемых в определенном порядке. Потоки позволяют одновременно выполнять несколько ядер или операции копирования памяти и запуск ядра.

Поток определяется созданием объекта потока и указанием последовательности запуска ядер или операций копирования на устройство в качестве параметра.

Каждая операция на устройстве привязана к потоку (по умолчанию – 0). Операции в одном потоке всегда выполняются последовательно. Операции в разных потоках могут быть выполнены параллельно.

Ниже приведены основные команды для работы с потоками.

```
// создание потока
cudaStream_t stream;
cudaStreamCreate(&stream);
// освобождение потока
cudaStreamDestroy(stream);
// асинхронное копирование, привязанное к потоку
cudaMemcpyAsync(dst, src, size, type, stream);
// вызов ядра, привязанный к потоку
kernel<<<grid, block, shared_mem, stream>>>(...);
```

В следующем коде приведен пример создания двух потоков и размещения массива hostPtr типа float в pinned-памяти.

```
cudaStream_t stream[];
for (int i = 0; i < 2; i++)
  cudaStreamCreate(&stream[i]);
float * hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

Каждый из этих потоков определяет последовательность одного копирования с CPU на GPU, одного запуска ядра и одного копирования с GPU на CPU (Рис. 42):

```
for (int i = 0; i < 2; ++i){
  cudaMemcpyAsync(inputDevPtr + i* size, hostPtr + i*size,
                  size, cudaMemcpyHostToDevice, stream[i]);
```

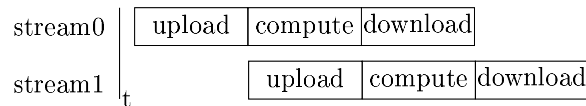

Рис. 42. Асинхронное копирование и выполнение ядра

```
  kernel<<<block, thread, 0, stream[i]>>>(outputDevPtr
                + i*size,inputDevPtr + i*size, size);
  cudaMemcpyAsync(hostPtr + i* size, outputDevPtr + i*size,
                size, cudaMemcpyDeviceToHost, stream[i]);
}
```

Каждый поток копирует свою часть входного массива `hostPtr` в массив `inputDevPtr` в памяти устройства, обрабатывает `inputDevPtr` на устройстве в ядре `kernel()` и копирует результат обратно в ту же часть массива `hostPtr`.

Общее время, затраченное на выполнение этого кода, вычисляется как: `total time = t(upload ) + t(download) + 2 * t(compute)`.

Потоки освобождаются вызовом функции `cudaStreamDestroy()`:

```
for (int i = 0; i < 2; i++)
  cudaStreamDestroy(stream[i]);
```

Функция `cudaStreamDestroy()` ждет, пока все предшествующие команды в данном потоке не завершены, затем уничтожает поток и передает управление CPU.

Одновременность выполнения возможна только с pinned-памятью. Одновременное выполнение ядра и копирования с/на устройство возможно, если установлено свойство устройства `deviceOverlap`.

Существуют различные способы явно синхронизировать потоки:

- `cudaThreadSynchronize()` ждет завершения всех операций во всех потоках;
- `cudaStreamSynchronize(stream)` ждет завершения всех операций в заданном потоке, остальные потоки могут работать на устройстве;
- `cudaStreamQuery(stream)` позволяет узнать приложению, завершены ли все операции в заданном потоке.

Существует также неявная синхронизация. Две команды из разных потоков не могут быть запущены одновременно, если одна из следующих команд запущена между ними нитью CPU:

- выделение pinned-памяти;
- выделение памяти на устройстве;
- установка значения в память устройства;
- копирование с устройства на устройство.

Некоторые устройства с cc=2.0 и выше (Fermi) могут выполнять несколько ядер одновременно. Приложение может узнать наличие этой возможности в свойстве устройства `concurrentKernels`. Максимальное число одновременных запусков ядер равно 16. Ядро из одного CUDA потока может выполняться одновременно с ядром из другого потока.

5.3 Занятость мультипроцессора

Инструкции нитей выполняются последовательно, поэтому выполнение других warp'ов – единственный способ скрыть латентность и загрузить видеокарту. *Занятость* (occupancy) – отношение числа warp'ов, вы-

полняемых одновременно на мультипроцессоре, к максимальному числу warp'ов, которые могут выполняться одновременно.

Мультипроцессор обладает ограниченным набором регистров и объемом разделяемой памяти:

- Число регистров на ядро:
  8K/16K/32K на мультипроцессор, разделены между потоками.
- Объем разделяемой памяти:
  16KB/48KB на мультипроцессор, разделены между потоками.

Компиляция с флагом `-ptxas-options=-v` предоставит информацию о количестве регистров, размере используемой локальной, разделяемой и константной памяти.

Лист Microsoft Excel Occupancy Calculator (Рис. 43) позволяет вычислить занятость GPU для заданного ядра. Калькулятор поможет выбрать размер блока, основываясь на информации о количестве разделяемой памяти и использовании регистров.

Для того чтобы правильно подобрать размер блока, потребуется проводить численные эксперименты, однако можно привести некоторые рекомендации:

- число нитей в блоке должно быть кратно размеру warp'а, чтобы избежать лишних вычислений на неполных warp'ах;
- в блоке должно быть минимум 64 нити при условии, что несколько блоков выполняются на одном мультипроцессоре;
- выбор между 128 и 256 нитями в блоке позволит задать первоначальный размер для численных экспериментов;

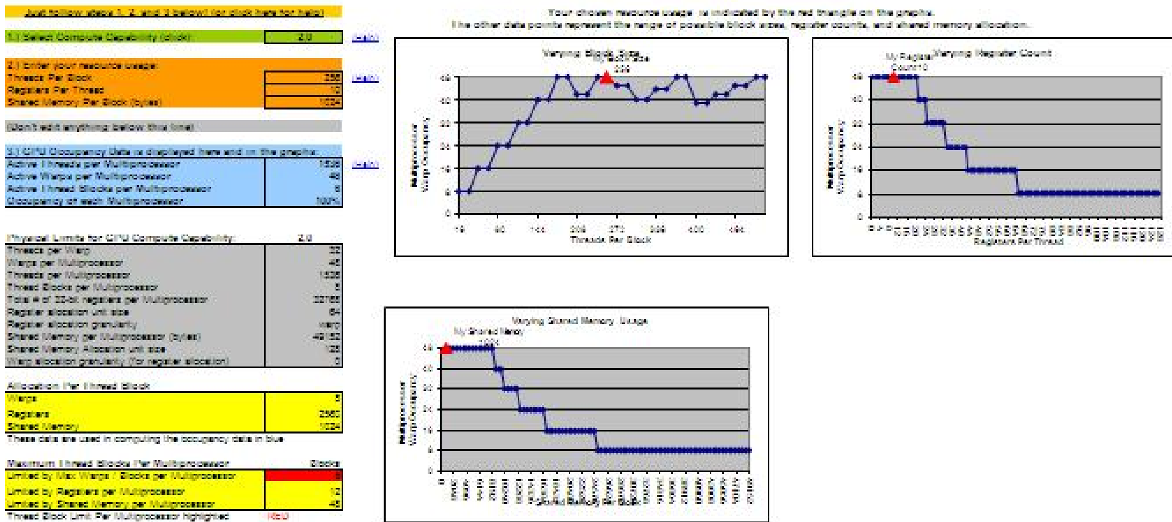

Рис. 43. Occupancy Calculator

- лучше использовать несколько небольших блоков, чем один большой блок на мультипроцессор, особенно если в ядре часто вызывается команда `__syncthreads()`.

5.4 Использование CUDA-профайлера

В составе CUDA tools идет Visual Profiler – средство для поиска узких мест в CUDA коде. Поддерживается на Linux / Windows / Mac платформах.

Профайлер при запуске требует указать приложение (путь до исполняемого файла) и выбрать счетчики (counters), которые нужно отслеживать. Каждый счетчик показывает, как часто происходит какое-то со-

бытие. После этого программа запускается, и профайлер регистрирует значения счетчиков по ходу ее исполнения для каждого ядра в отдельности.

Когда выполнение программы завершено, профайлер составляет таблицу на каждый запуск ядер, в которой можно посмотреть значения счетчиков и выполнить анализ узких мест в коде. Также профайлер обрабатывает собранные данные и представляет их в виде отчетов (Рис. 44–48). Можно узнать такую информацию как время, затраченное на выполнение каждого ядра, количество обращений к памяти, количество запущенных блоков, количество расходящихся ветвей внутри warp'a и так далее.

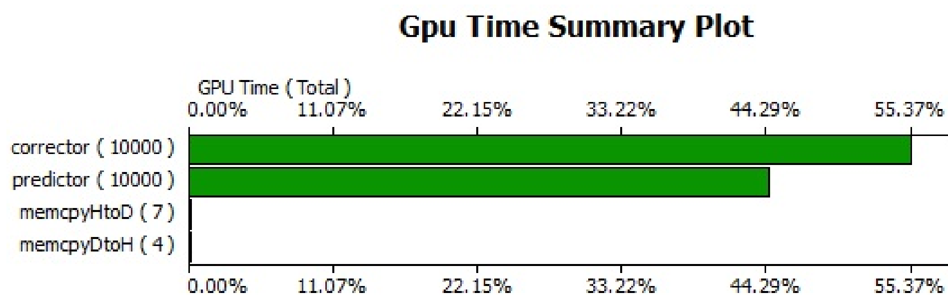

Рис. 44. Итоговый график времени выполнения

Session1::Device_0::Context_0

- Kernel time = 2.34 % of total GPU time
- Memory copy time = 0.0 % of total GPU time
- Kernel taking maximum time = **corrector** (1.3% of total GPU time)
- Memory copy taking maximum time = **memcpyHtoD** (0.0% of total GPU t

Рис. 45. Информация по работе программы

	GPU Timestamp (us)	Method	GPU Time (us)	CPU Time (us)	Occupancy	grid size	thread block size	registers
1	0	memcpy...	1,184	1,916				
2	99,584	memcpy...	1,152	1,532				
3	718,592	memcpy...	1,568	2,717				
4	1511,94	memcpy...	88,864	247,112				
5	1847,81	memcpy...	88,8	241,748				
6	2185,22	memcpy...	88,832	243,28				
7	2521,34	memcpy...	88,8	242,131				
8	2751,49	predictor	17,216	129,494	1	[140 1]	[512 1 1]	11
9	3920,13	corrector	20,256	125,663	1	[140 1]	[512 1 1]	8
10	4769,79	predictor	16,16	119,533	1	[140 1]	[512 1 1]	11
11	5659,9	corrector	20,672	124,897	1	[140 1]	[512 1 1]	8
12	6507,26	predictor	16,192	119,916	1	[140 1]	[512 1 1]	11
13	7349,76	corrector	20,832	124,514	1	[140 1]	[512 1 1]	8
14	8198,14	predictor	16,32	120,299	1	[140 1]	[512 1 1]	11

Рис. 46. Итоговая таблица

	GPU Timestamp (us)	Method	dram reads Type:FB	dram writes Type:FB	L1 cache global hit ratio (%)	L1 cache read throughput(GB/s)
1	99,584	memcpy...				
2	0	memcpy...				
3	718,592	memcpy...				
4	588020	predictor	24267	21247	75,1127	206,607
5	4,85239e+06	predictor	24083	21371	74,3439	219,09
6	2,43745e+06	predictor	23703	21366	77,424	221,182
7	5,50516e+06	predictor	23757	21404	77,5854	209,434
8	1,57833e+07	predictor	23887	21408	75,9629	217,111
9	1,26676e+07	predictor	23845	21409	76,8218	219,188
10	5,33374e+06	predictor	23913	21441	76,7672	218,856
11	1,26343e+07	predictor	24169	21448	72,6594	174,582
12	6,58655e+06	predictor	24193	21420	77,3314	210,587
13	1,54168e+07	corrector	48953	20313	46,0413	100,985
14	4,02029e+06	corrector	49011	20266	41,3164	99,9467

Рис. 47. Вычисленные данные для ядер.

	Method	#Calls	mem transfer size (bytes)	host mem transfer type
1	memcpyDtoH	2	286720	Pageable
2	memcpyDtoH	2	286724	Pageable
3	memcpyHtoD	3	4	Pageable
4	memcpyHtoD	2	286720	Pageable
5	memcpyHtoD	2	286724	Pageable

Рис. 48. Таблица копирования памяти

Профилирование кода может быть запущено вручную, без использования Visual Profiler. Для этого нужно установить переменную среды `CUDA_PROFILE=1`. Информация о профилировании будет записана в лог-файл.

Дополнительные опции:

- `CUDA_PROFILE_CSV=1` – переключение режима для CSV формата, импорта в Visual Profiler;
- `CUDA_PROFILE_LOG=<filename>` – явно задает имя лог файла;
- `CUDA_PROFILE_CONFIG=<filename>` – задает имя файла конфигурации.

5.5 Использование отладчиков и PTX-ассемблера

Для отладки параллельного кода в Windows существует программа NVIDIA Parallel Nsight. Эта программа интегрируется с Microsoft Visual Studio. При создании проекта в Visual Studio можно выбрать раздел

NVIDIA CUDA со всеми настройками, необходимыми для работы. Основные возможности NVIDIA Parallel Nsight:

- отладка ядер непосредственно на GPU;
- мониторинг параллельных потоков и памяти;
- обнаружение и корректировка ошибок в массивно-параллельном коде при помощи условных точек остановки.

CUDA-GDB – инструмент отладки CUDA приложений, работающих на платформе Linux. NVIDIA дополнила отладчик gdb, реализовав возможность выполнять отладку кода на устройстве. Основные возможности отладчика CUDA-GDB:

- установка контрольных точек в исходном коде на CUDA C;
- просмотр глобальной и разделяемой памяти;
- перечисление блоков и нитей, исполняемых на GPU;
- пошаговое выполнение warp'а нитей.

Для более тонкой отладки кода можно воспользоваться анализом PTX-ассемблера. PTX – Portable Thread Execution – виртуальная машина и система команд.

Ключи `-keep` или `-ptx` позволяют получить промежуточный ptx-ассемблер.

Просмотр PTX-кода может выявить проблемы производительности, так как компилятор CUDA nvcc производит не всегда оптимальный код.

Вопросы и задания

1. Приведите примеры оптимизации вычислений математических функций.
2. Для чего используется поток?
3. С помощью каких функций происходит создание и уничтожение потоков?
4. Как реализовать синхронизацию потоков?
5. При каких условиях возможно одновременное копирование памяти и выполнение ядра?
6. При каких условиях возможно одновременное выполнение нескольких ядер?
7. Как вычислить занятость мультипроцессора?
8. Какие существуют рекомендации по выбору размера блока?
9. Какие средства отладки приложений на CUDA вы знаете?
10. Для чего используется Visual Profiler?
11. Проведите анализ работы программ из упражнений к главе 4 с помощью CUDA-профайлера.

6 Работа с несколькими устройствами

6.1 Получение информации об имеющихся GPU

В системе может быть несколько устройств, поддерживающих технологию CUDA. Прежде чем начать работу с несколькими GPU, важно получить информацию обо всех имеющихся GPU и их возможностях. CUDA runtime API предоставляет такую информацию в виде структуры `cudaDeviceprop`.

Ниже приводится исходный текст программы, перечисляющий все доступные GPU и их основные возможности.

```
int main( int argc, const char** argv){
  int deviceCount = 0;
  cudaGetDeviceCount(&deviceCount);
  if (deviceCount == 0)
    printf("There is no device supporting CUDA\n");
  else if (deviceCount == 1)
    printf("There is 1 device supporting CUDA\n");
  else
    printf("There are %d devices supporting CUDA\n", deviceCount);

  for (int dev = 0; dev < deviceCount; ++dev){
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, dev);

    printf("\nDevice %d: \"%s\"\n", dev, deviceProp.name);
    printf("CUDA Capability Major/Minor version number:   %d.%d\n",
           deviceProp.major, deviceProp.minor);
    printf("Amount of global memory:                     %d bytes\n",
           deviceProp.totalGlobalMem);
```

```
    printf("Amount of constant memory:           %d bytes\n",
           deviceProp.totalConstMem);
    printf("Amount of shared memory per block:    %d bytes\n",
           deviceProp.sharedMemPerBlock);
    printf("Number of registers per block: %d\n",
           deviceProp.regsPerBlock);
    printf("Warp size:                             %d\n",
           deviceProp.warpSize);
    printf("Maximum number of threads per block:   %d\n",
           deviceProp.maxThreadsPerBlock);
    printf("Maximum thread dim:      %d x %d x %d\n",
           deviceProp.maxThreadsDim[0],
           deviceProp.maxThreadsDim[1],
           deviceProp.maxThreadsDim[2]);
    printf("Maximum grid size:        %d x %d x %d\n",
           deviceProp.maxGridSize[0],
           deviceProp.maxGridSize[1],
           deviceProp.maxGridSize[2]);
    printf("Clock rate:                        %.2f GHz\n",
           deviceProp.clockRate * 1e-6f);
   }
 return 0;
}
```

6.2 Выбор устройства

CPU может управлять любым устройством при помощи вызова функции `cudaSetDevice()`. Выделение памяти на устройстве и запуск ядер происходит на текущем установленном устройстве. Потоки и события также привязаны к устройству.

Если не было вызова функции cudaSetDevice(), текущим устройством по умолчанию является устройство 0.

Следующий код показывает, как установка текущего устройства влияет на выделение памяти и вызов ядра.

```
size_t size = 1024 * sizeof(float);
cudaSetDevice(0); // установить текущим устройство 0
float * p0;
cudaMalloc(&p0, size); // выделить память на устройстве 0
kernel<<<block, thread>>>(p0);// вызов ядра на устройстве 0
cudaSetDevice(1); // установить текущим устройство 1
float * p1;
cudaMalloc(&p1,size); // выделить память на устройстве 1
kernel<<<block, thread>>>(p1);// вызов ядра на устройстве 1
```

Замечание: cudaDeviceSynchronize() ждет завершения операций только на текущем устройстве.

6.3 Потоки и события

Запуск ядра или копирование памяти вызовет ошибку, если оно запущено в потоке, не привязанном к текущему устройству, как показано в следующем коде:

```
cudaSetDevice(0); // установить текущим устройство 0
cudaStream_t s0;
cudaStreamCreate(&s0); // создать поток s0 на устройстве 0
kernel<<<block, thread, 0, s0>>>(); // устройство 0, s0
cudaSetDevice(1); // установить текущим устройство 1
cudaStream_t s1;
cudaStreamCreate(&s1); // создать поток s1 на устройстве 1
```

```
kernel<<<block, thread, 0, s1>>>(); // устройство 1, s1
// этот запуск ядра вызовет ошибку:
kernel<<<block, thread, 0, s0>>>(); // устройство 1, s0
```

Замечания:

- `cudaEventRecord()` вызовет ошибку, если событие и поток привязаны к разным устройствам;
- `cudaEventElapsedTime()` вызовет ошибку, если два потока привязаны к разным устройствам;
- `cudaEventSynchronize()` не вызовет ошибку, даже если событие привязано не к текущему устройству.

6.4 Единое адресное пространство

Если приложение выполняется на 64-рязрядной системе, все типы памяти, к которым устройство имеет доступ, располагаются в одном адресном пространстве. На рисунках 49 и 50 схематически показано распределение памяти CPU и нескольких GPU.

Единое адресное пространство используется для всех устройств с версией cc=2.0 и выше (Fermi). Это адресное пространство применяется для всей памяти, выделенной CPU с помощью функции `cudaHostAlloc()`, и всей памяти, выделенной на любом из устройств с помощью функции `cudaMalloc*()`. Определить на какую память – CPU или GPU – ссылается указатель можно с помощью вызова `cudaPointGetAttributes()`.

При копировании в память или из памяти устройства, для которого используется единое адресное пространство, параметр `cudaMemcpyKind`

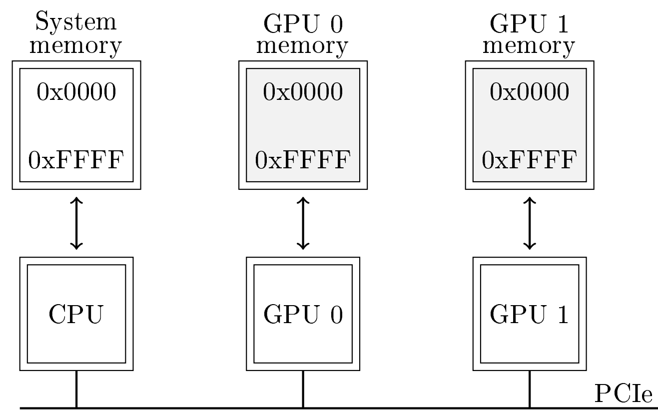

Рис. 49. Несколько адресных пространств

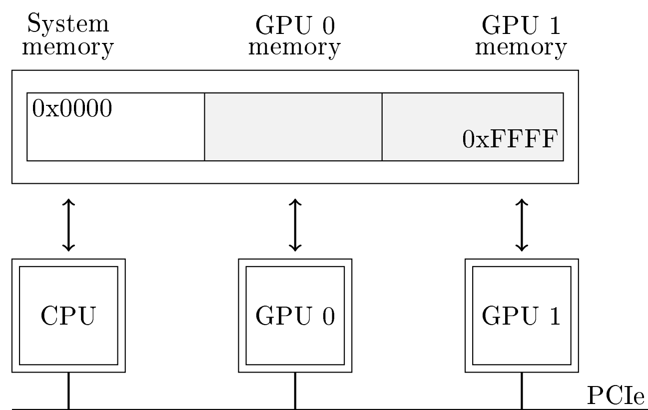

Рис. 50. Единое адресное пространство

в функции `cudaMalloc*()` становится ненужным, и может быть установлен как `cudaMemcpyDefault`.

Приложение может проверить используется ли для конкретного устройства единое адресное пространство запросом свойства `unifiedAddressing`.

6.5 Копирование с устройства на устройство

Если приложение выполняется на 64-рязрядной системе, устройства с cc=2.0 и выше (Fermi) могут обращаться к памяти друг друга (то есть ядро, выполняемое на одном устройстве, может обращаться к памяти другого устройства). Такой доступ к памяти (*peer-to-peer memory access*) поддерживается между двумя устройствами, если оба устройства имеют свойство `cudaDeviceCanAccessPeer`. Также нужно вызвать функцию `cudaDeviceEnablePeerAccess()`, как показано в следующем коде. Единое адресное пространство используется для двух устройств, поэтому один и тот же указатель может использоваться двумя устройствами.

```
cudaSetDevice(0); // установить текущим устройство 0
float * p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size); // выделить память на устройстве 0
kernel<<<1000, 128>>>(p0); // вызов ядра на устройстве 0
cudaSetDevice(1); // установить текущим устройство 1
cudaDeviceEnablePeerAccess(0, 0); // разрешить peer-to-peer
                                  // доступ к устройству 0
// запустить ядро на устройстве 1
// это ядро может использовать память устройства 0 по адресу p0
kernel<<<thread, block>>>(p0);
```

Если единое адресное пространство не используется, то копирование данных между двумя устройствами можно выполнить при помощи функций `cudaMemcpyPeer()`, `cudaMemcpyPeerAsync()`, как показано в следующем коде.

```
cudaSetDevice(0); // установить текущим устройство 0
float * p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size); // выделить память на устройстве 0
cudaSetDevice(1); // установить текущим устройство 1
float * p1;
cudaMalloc(&p1, size); // выделить память на устройстве 1
cudaSetDevice(0); // установить текущим устройство 0
kernel<<<block, thread>>>(p0); // вызов ядра на устройстве 0
cudaSetDevice(1); // установить текущим устройство 1
cudaMemcpyPeer(p1, 1, p0, 0, size); // копировать p0 в p1
kernel<<<block, thread>>>(p1); // вызов ядра на устройстве 1
```

Копирование памяти между двумя разными устройствами не начинается, пока не завершены все команды, запущенные ранее на одном из устройств.

Если устройства имеют свойство `cudaDeviceCanAccessPeer`, копирование памяти между этими устройствам происходит без участия CPU и, следовательно, является быстрым.

6.6 Декомпозиция области

Рассмотрим случай, когда требуется распределить данные по двум устройствам. Случай распределения данных на несколько устройств легко обобщается.

Разобьем вычислительную область поровну на два устройства, и добавим к каждому устройству часть области, необходимую для работы с данными на границе разбиения (Рис. 51). Каждое устройство обрабаты-

вает свою часть результата, получая обновленные данные с граничных нитей от другого устройства.

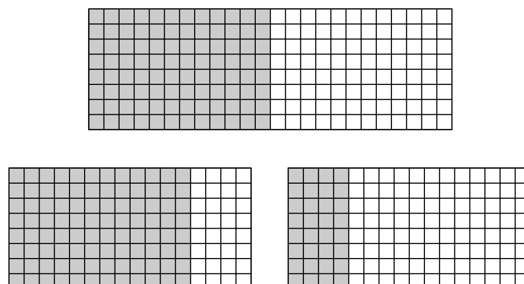

Рис. 51. Декомпозиция области

Для того, чтобы увеличить скорость выполнения работы, совместим обмен граничными нитями с выполнением ядра. Каждый шаг по времени выполняется в 2 этапа, как показано на рисунке 52.

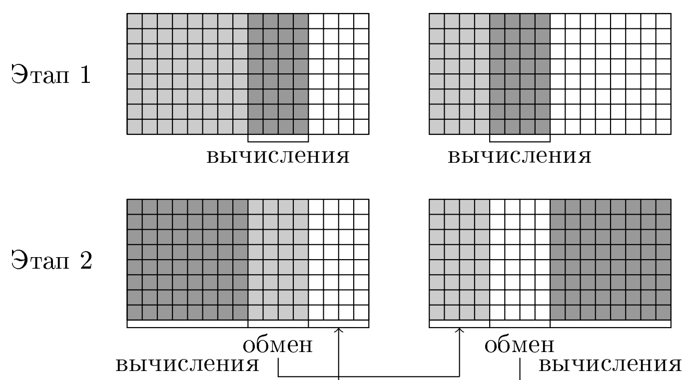

Рис. 52. Этапы вычисления одного шага по времени на 2 GPU

На первом этапе GPU вычисляет часть данных, соответствующих граничным нитям. На втором этапе GPU вычисляет ядро с оставшимися данными и одновременно обменивается данными с граничных нитей. CU-

DA обеспечивает асинхронное выполнение ядра и копирование памяти, позволяя обоим GPU обрабатывать данные во время копирования.

6.7 Использование OpenMP

Вариантом работы с несколькими устройствами GPU является использование потоков OpenMP. Каждый поток работает только на одном устройстве. Ниже представлен пример работы с двумя устройствами.

```
#pragma omp parallel {
  int thread = omp_get_thread_num();
  if (thread == 0){
    cudaSetDevice(0);
    ... // код для устройства 0
    cudaThreadSynchronize();
  } else if (thread == 1){
    cudaSetDevice(1);
    ... // код для устройства 1
    cudaThreadSynchronize();
}}
```

6.8 Использование MPI

Если требуется использовать несколько GPU, которые находятся на разных узлах CPU, можно воспользоваться библиотекой передачи сообщений MPI (Message Passing Interface). Принцип работы с библиотекой заключается в передаче сообщений между несколькими CPU в кластере. Для обмена сообщениями между устройствами потребуются 3 переда-

чи данных, как показано на рисунке 53. CPU обмениваются данными с помощью функции `MPI_sendrecv()`.

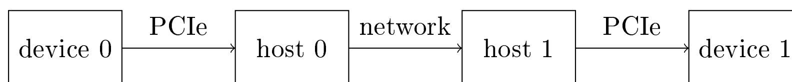

Рис. 53. CUDA+MPI

Для компиляции программы MPI+CUDA потребуется использование компиляторов mpicc и nvcc.

Вопросы и задания

1. С помощью какой функции осуществляется выбор текущего устройства?
2. Как запустить ядро, привязанное к потоку?
3. Что понимается под единым адресным пространством?
4. При каких условиях возможно копирование памяти с устройства на устройство напрямую?
5. С помощью какой функции можно копировать память с устройства на устройство напрямую?
6. Назовите подход к декомпозиции области на два устройства. Обобщите его на случай многих устройств.
7. Исследуйте способ совмещения CUDA+OpenMP.
8. Исследуйте способ совмещения CUDA+MPI.

7 Библиотеки CUDA

7.1 CUFFT

Библиотека CUFFT (CUDA-вариант библиотеки FFT, Fast Fourier Transform) разработана для расчета быстрого преобразования Фурье, широко используемого в задачах вычислительной физики и обработке сигналов. Библиотека поддерживает одномерное, двумерное или трехмерное преобразование Фурье вещественных и комплексных чисел одинарной и двойной точности.

Формулы для одномерного прямого и обратного преобразования Фурье имеют вид:

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi i}{N} kn}, \quad k = 0, ..., N-1$$

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N} kn}, \quad n = 0, ..., N-1$$

Для того чтобы выполнить преобразование Фурье, требуется создать план для хранения конфигурации размера исходного сигнала и типа данных:

```
cufftHandle plan;
```

В следующем коде приведены команды для создания плана одномерного, двумерного или трехмерного преобразования Фурье.

```
cufftResult cufftPlan1d(cufftHandle *plan,
            int nx, cufftType type, int batch);
cufftResult cufftPlan2d(cufftHandle *plan,
            int nx, int ny, cufftType type);
cufftResult cufftPlan3d(cufftHandle *plan,
            int nx, int ny, int nz, cufftType type);
```

Параметры nx, ny, nz содержат информацию о размере сигнала, type – тип сигнала. Параметр batch определяет число одномерных преобразований.

Преобразование Фурье выполняется согласно плану с помощью следующих функций:

```
cufftResult cufftExecC2C(cufftHandle plan, cufftComplex * idata,
       cufftComplex * odata, int direction);
cufftResult cufftExecR2C(cufftHandle plan, cufftReal * idata,
       cufftComplex * odata);
cufftResult cufftExecC2R(cufftHandle plan, cufftComplex * idata,
       cufftReal * odata);
cufftResult cufftExecZ2Z(cufftHandle plan, cufftDoubleComplex *

       idata, cufftDoubleComplex * odata, int direction);
```

Параметры idata, odata содержат указатели на входные и выходные данные в памяти GPU. Параметр направления direction определяет тип преобразования Фурье:

- CUFFT_FORWARD – прямое преобразование Фурье;

- `CUFFT_INVERSE` – обратное преобразование Фурье.

Для завершения работы библиотеки требуется вызвать команду для уничтожения плана:

```
cufftResult cufftDestroy(cufftHandle plan);
```

В следующем коде приводится пример одномерного преобразования Фурье Complex-to-Complex.

```
#define NX 256
#define BATCH 10
cufftHandle plan;
cufftComplex * idata, * odata;
cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*BATCH);
cudaMalloc((void**)&odata, sizeof(cufftComplex)*NX*BATCH);
// создать 1D план
cufftPlan1d(&plan, NX, CUFFT_C2C, BATCH);
// использовать план для преобразования сигнала
cufftExecC2C(plan, idata, odata, CUFFT_FORWARD);
// обратное преобразование сигнала
cufftExecC2C(plan, odata, odata, CUFFT_INVERSE);
// уничтожить план
cufftDestroy(plan);
cudaFree(idata);
cudaFree(odata);
```

В этом примере, для того чтобы получить исходный сигнал, требуется разделить значения выходного сигнала на число элементов сигнала.

Для работы с библиотекой в код нужно добавить

```
#include <cufft.h>
```

Если проект написан по типовому шаблону SDK, в Makefile также нужно добавить строчку

```
USECUFFT := 1
```

7.2 CUBLAS

Библиотека CUBLAS (CUDA-вариант библиотеки BLAS, Basic Linear Algebra Subprograms) предназначена для решения задач линейной алгебры с использованием прямого доступа к ресурсам GPU.

Для того чтобы воспользоваться библиотекой в приложении, нужно разместить в памяти устройства требуемые матрицы и векторы, заполнить их значениями, вызвать CUBLAS функции и загрузить результат в CPU.

Для инициализации CUBLAS требуется обработчик для вызова всех функций в библиотеке.

```
cublasHandle_t cublasHandle;
cublasStatus_t status = cublasCreate(&cublasHandle);
...
cublasDestroy(cublasHandle);
```

Все функции для математических операций имеют вид:

```
cublasStatus_t cublas<T><operation>(cublasHandle_t handle,...);
```

Буква T обозначает тип данных:

- S = `float`;
- D = `double`;
- C = `complex float`;
- Z = `complex double`.

В следующем коде приводится пример вычисления нормы вектора типа `float`:

```
cublasStatus_t cublasSnrm2(cublasHandle_t handle, int n,
    const float * x, int incX, float * result);
```

Параметр `incX` – инкремент, шаг для хранения элементов массивов по частям.

Далее приводятся команды вызова различных операций линейной алгебры.

- Норма вектора $\sqrt{\sum_k x_k^2}$:

  ```
  cublasStatus_t cublas<T>nrm2(cublasHandle_t handle, int n,
    const T * x, int incX, T * result);
  ```

- Сумма абсолютных значений вектора $\sqrt{\sum_k |x_k|}$:

  ```
  cublasStatus_t cublas<T>asum(cublasHandle_t handle, int n,
    const T * x, int incX, T * result);
  ```

- Скалярное произведение векторов $\sqrt{\sum\limits_k x_k y_k}$:

```
cublasStatus_t cublas<T>dot(cublasHandle_t handle, int n,
 const T * x, int incX, const T * y, int incY, T * result);
```

- Сложение векторов с умножением на число $y = \alpha x + y$:

```
cublasStatus_t cublas<T>axpy(cublasHandle_t handle, int n,
 const T * alpha, const T * x, int incX, T * y, int incY);
```

- Умножение матрицы на вектор $y = \alpha Ax + \beta y$:

```
cublasStatus_t cublas<T>gemv(cublasHandle_t handle,
 cublasOperation_t transpose, int m, int n,
 const T * alpha, const T * A, int pitchA,
 const T * x, int incX, const T * beta, T * y, int incY);
```

- Умножение матрицы на матрицу $C = \alpha AB + \beta C$:

```
cublasStatus_t cublas<T>gemm(cublasHandle_t handle,
 cublasOperation_t transposeA, cublasOperation_t transposeB,
 int m, int n, int k, const T * alpha,
 const T * A, int pitchA, const T * B, int pitchB,
 const T * beta, T * C, int pitchC);
```

Перечислим некоторые способы хранения матриц:

- ge (general matrix) – обычная;
- gb (general band matrix) – ленточная;
- sy (symmetric matrix) – симметричная;
- he (Hermitian matrix) – эрмитова;
- tr (triangular matrix) – треугольная.

Для именования функций могут использоваться:

- mv – матрично-векторное умножение;
- sv – решение СЛАУ с операцией матрично-векторного умножения;
- mm – умножение матриц;
- sm – решение СЛАУ с операцией умножения матриц.

Библиотека ориентирована на Фортран. Поэтому в целях совместимости с существующими версиями:

- в матрицах последовательно лежат элементы в столбцах;
- нумерация элементов начинается с единицы;
- проверка ошибок производится отдельными функциями.

Для C/C++ при вычислении индексов массивов рекомендуется использовать макросы:

```
#define IDX2F(i,j,ld) ((((j)-1)*(ld))+((i)-1))
#define IDX2C(i,j,ld) (((j)*(ld))+(i))
```

Для работы с библиотекой в код нужно добавить

```
#include <cublas.h>
```

Если проект написан по типовому шаблону SDK, в Makefile также нужно добавить строчку

```
USECUBLAS := 1
```

7.3 CUSPARSE

Библиотека CUSPARSE содержит набор процедур линейной алгебры и используется для работы с разреженными матрицами. Эти процедуры могут быть классифицированы по 4 категориям:

- уровень 1 – операции между разреженными и «плотными» векторами;
- уровень 2 – операции между разреженными матрицами и «плотными» векторами;
- уровень 3 – операции над разреженными матрицами и множеством «плотных» векторов;
- вспомогательные функции и функции преобразования типов.

Библиотека работает с матрицами, размещенными в памяти GPU.

Для инициализации CUSPARSE требуется обработчик для вызова всех функций в библиотеке.

```
cusparseHandle_t cusparseHandle;
cusparseStatus_t status = cusparseCreate(&cusparseHandle);
...
cusparseDestroy(cusparseHandle);
```

CUSPARSE поддерживает формат хранения разреженных векторов и форматы разреженных матриц:

- *диагональный* (DIA);
- *ELLPACK* (ELL);
- *координатный* (COO);

- *сжатый по строкам* (CSR).

Формат хранения разреженных векторов

Разреженные векторы хранятся в двух массивах. Первый массив содержит ненулевые значения массива, второй массив – массив индексов – хранит позицию соответствующего ненулевого элемента.

Например, вектор

$$(\ 1.0 \ \ 0.0 \ \ 0.0 \ \ 3.0 \ \ 4.0 \ \ 0.0 \ \ 7.0 \ )$$

может быть записан в виде

$$\begin{pmatrix} 1.0 & 3.0 & 4.0 & 7.0 \\ 0 & 3 & 4 & 6 \end{pmatrix}.$$

Предполагается, что индексы расположены в возрастающем порядке и не повторяются.

Диагональный формат

Диагональный формат формируется из двух массивов. Массив $data$ содержит ненулевые значения, массив $offsets$ хранит отступы каждой диагонали от главной диагонали:
$i = 0$ – главная диагональ,
$i > 0$ – i-я верхняя диагональ,
$i < 0$ – i-я нижняя диагональ.

$$A = \begin{pmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{pmatrix}, \quad data = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix}, \quad offsets = \begin{bmatrix} -2 & 0 & 1 \end{bmatrix}.$$

Значения $*$ могут содержать произвольный символ.

На рисунке 54 приводится пример матрицы, хорошо описываемой диагональным форматом. Стоит отметить, что многие разреженные матрицы имеют структуру, не подходящую для диагонального формата (Рис. 55).

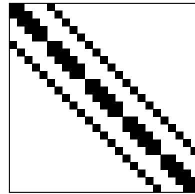

Рис. 54. Пример матрицы, описываемой диагональным форматом. Разреженная матрица 25×25

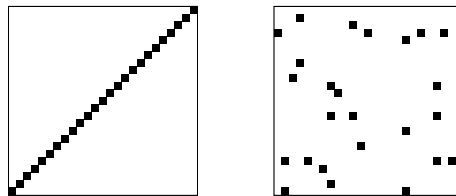

Рис. 55. Примеры матриц, плохо описываемых диагональным форматом

ELLPACK формат

Для $M \times N$ матрицы, в которой максимум K элементов в строке ненулевые, ELLPACK формат хранит ненулевые элементы в плотной матрице $data$ $M \times K$. Номера столбцов хранятся в $indices$. Этот формат является более общим, чем диагональный. Он подходит, когда максимальное число ненулевых элементов в строке незначительно отличается от среднего.

$$A = \begin{pmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{pmatrix}, \quad data = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix}, \quad indices = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix}.$$

Координатный формат

Координатный формат представляет собой простую схему хранения данных. Массивы row, col и $data$ содержат номера строки, столбца и ненулевые значения соответственно. Координатный формат является общим для представления разреженных матриц, так как для произвольной разреженной матрицы число элементов, которое требуется хранить, пропорционально числу ненулевых элементов. В отличие от диагонального и ELL формата, в координатном формате индексы строки и столбца хранятся явно.

$$A = \begin{pmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{pmatrix}$$

$$row = [\ 0 \quad 0 \quad 1 \quad 1 \quad 2 \quad 2 \quad 2 \quad 3 \quad 3\ ]$$
$$col = [\ 0 \quad 1 \quad 1 \quad 2 \quad 0 \quad 2 \quad 3 \quad 1 \quad 3\ ]$$
$$data = [\ 1 \quad 7 \quad 2 \quad 8 \quad 5 \quad 3 \quad 9 \quad 6 \quad 4\ ]$$

CSR формат

Сжатый по строкам формат (CSR, Compressed sparse row), так же как и координатный формат, явно хранит номера столбцов и ненулевые элементы в массивах *indices* и *data*. Массив указателей *ptr* отвечает за CSR представление матрицы.

Для матрицы $M \times N$ массив *ptr* имеет длину $M+1$ и хранит отступы i-й строки в $ptr[i]$. Последнее значение в *ptr* хранит число ненулевых элементов в матрице.

$$A = \begin{pmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{pmatrix}$$

$$ptr = [\ 0\ \ 2\ \ 4\ \ 7\ \ 9\ ]$$

$$col = [\ 0\ \ 1\ \ 1\ \ 2\ \ 0\ \ 2\ \ 3\ \ 1\ \ 3\ ]$$

$$data = [\ 1\ \ 7\ \ 2\ \ 8\ \ 5\ \ 3\ \ 9\ \ 6\ \ 4\ ]$$

Ненулевые элементы хранятся построчно в массиве *data*. Массив индексов *indices* хранит позицию элемента в столбце исходной матрицы. Массив указателей *ptr* содержит указатели на элементы матрицы. Вторая строка начинается со второго элемента, третья строка – с четвертого, четвертая строка – с седьмого.

CUSPARSE функции

Рассмотрим для примера вызов функции умножения разреженной матрицы, хранящейся в CSR формате, на вектор:

$$y = \alpha Ax + \beta y$$

```
cusparseStatus_t cusparse<T>csrmv(cusparseHandle_t handle,
 cusparseOperation_t transpose, int m, int n,
 const T * alpha, const cusparseMatDescr_t descrA,
 const T * data, const int * ptr,
 const int * col,
 const T * x, const T * beta, T * y);
```

Разреженные матрицы описываются структурой данных matrix description. Поля структуры `cusparseMatDescr_t` описывают детали хра-

нения матрицы: симметричная, эрмиттова, треугольная, индексация с 0 или 1.

Функция cusparseCreateMatDescr() инициализирует несимметричную матрицу с индексацией от 0.

```
cusparseStatus_t cusparseCreateMatDescr(cusparseMatDescr_t * descrA);
```

Для уничтожения структуры данных вызывается функция:

```
cusparseStatus_t cusparseDestroyMatDescr(cusparseMatDescr_t descrA);
```

Для работы с библиотекой в код нужно добавить

```
#include <cusparse.h>
```

Если проект написан по типовому шаблону SDK, в Makefile также нужно добавить строчку

```
USECUSPARSE := 1
```

7.4 CURAND

Библиотека CURAND предназначена для генерации псевдослучайных чисел. CURAND представлена в двух частях: Host API и Device API. В Host API функции вызываются из CPU и выполняются на GPU.

Сгенерированная последовательность случайных чисел хранится в глобальной памяти устройства. Device API позволяет вызывать функции генерации случайных чисел непосредственно из ядра.

Последовательность работы с библиотекой в Host API:

1. Создать генератор `curandCreateGenerator()` желаемого типа
   ```
   curandStatus_t curandCreateGenerator(curandGenerator_t *
                  generator, curandRngType_t type)
   ```
   Тип определяет метод генерации случайных чисел.
2. Задать параметры генератора, например, `curandSetPseudoRandomGeneratorSeed()`.
3. Выделить память на устройстве с помощью функции `cudaMalloc()`.
4. Сгенерировать последовательность случайных чисел с помощью функции `curandGenerate()`:

   ```
   curandStatus_t curandGenerate(curandGenerator_t generator,
         unsigned int * outputPtr, size_t num);
   curandStatus_t curandGenerateUniform(curandGenerator_t generator,
         float * outputPtr, size_t num);
   curandStatus_t curandGenerateUniformDouble(curandGenerator_t
         generator, double * outputPtr, size_t num);
   curandStatus_t curandGenerateNormal(curandGenerator_t
         generator, float * outputPtr, size_t num,
         float mean, float stddev);
   curandStatus_t curandGenerateNormalDouble(curandGenerator_t
         generator, double * outputPtr, size_t num,
         double mean, double stddev);
   ```

5. Использовать результаты.
6. При необходимости повторить вызов функции `curandGenerate()`.
7. Уничтожить генератор

```
curandStatus_t curandDestroyGenerator(curandGenerator_t
               generator);
```

В device API перед началом работы требуется инициализация состояния `state`:

```
__device__ void curand_init(unsigned long long seed,
                           unsigned long long sequence,
                           unsigned long long offset,
                           curandState_t *state)
```

Для генерации случайных чисел используется функция `curand()`. Если `curand()` вызывается с одним и тем же состоянием, будет сгенерирована та же самая последовательность. При генерации случайных чисел параллельно множеством нитей, нити должны иметь различную структуру данных `state`. Параметр `seed` отвечает за генерацию различных состояний.

Функции для генерации случайных чисел в коде устройства:

```
__device__ unsigned int curand(curandState_t *state)
__device__ float curand_uniform(curandState_t *state)
__device__ double curand_uniform_double(curandState_t *state)
__device__ float curand_normal(curandState_t *state)
__device__ double curand_normal_double(curandState_t *state)
```

Для работы с библиотекой в код нужно добавить

```
#include <curand.h>
#include <curand_ kernel.h>
```

Если проект написан по типовому шаблону SDK, в Makefile также нужно добавить строчку

```
USECURAND := 1
```

7.5 Потоки

Библиотеки позволяют задавать потоки обработчика:

```
cufftResult cufftSetStream(cufftHandle plan,
                cudaStream_t stream);
cublasStatus_t cublasSetStream(cublasHandle_t handle,
                cudaStream_t stream);
cusparseStatus_t cusparseSetKernelStream(cusparseHandle_t handle,
                cudaStream_t stream);
curandStatus_t curandSetStream(curandGenerator_t generator,
                cudaStream_t stream);
```

Если ядра небольшие, задание потока позволяет запускать одновременно несколько ядер на Fermi.

Вопросы и задания

1. Напишите программу, реализующую преобразование Фурье для заданной функции с помощью библиотеки CUFFT.
2. Напишите программу, реализующую умножение матриц с помощью библиотеки CUBLAS.

3. Напишите программу, реализующую умножение матриц с помощью библиотеки CUSPARSE. Матрицы хранятся в координатном формате.
4. Напишите программу, реализующую умножение матриц с помощью библиотеки CUSPARSE. Матрицы хранятся в диагональном формате.
5. Напишите программу, реализующую умножение матриц с помощью библиотеки CUSPARSE. Матрицы хранятся в CSR формате.

Темы проектов

Для выполнения проектов по программированию и моделированию вам нужно:

- ознакомиться с литературой по теме;
- построить математическую модель задачи;
- построить алгоритм решения задачи;
- предложить (выполнить) программную реализацию алгоритма;
- описать алгоритм и процесс его построения в 10-минутном докладе.

Список проектов

1. Численное решение СЛАУ методом сопряженных градиентов.
   (a) Реализация на CUDA.
   (b) Реализация с использованием библиотеки CUBLAS.
   (c) Реализация с использованием библиотеки CUSPARSE. Координатный формат хранения данных.
   (d) Реализация с использованием библиотеки CUSPARSE. Сжатый по строкам формат хранения данных.
2. Численное решение трехдиагональных СЛАУ. Реализация метода параллельной циклической редукции.
3. Численное решение СЛАУ методом Якоби.
4. Численное решение СЛАУ методом простых итераций.
5. Реализация степенного метода нахождения максимального по модулю собственного числа.

6. Численное решение задачи Дирихле для уравнения Пуассона. Метод Гаусса-Зейделя.
7. Численное решение волнового уравнения.
   (a) Метод Годунова.
   (b) Метод «крест» Неймана-Рихтмайера.
8. Численное решение уравнения теплопроводности.
   (a) Метод Ричардсона.
   (b) Метод Кранка-Николсон.

Библиографический список

1. Боресков А.В., Харламов А.А. Основы работы с технологией CUDA. М.: ДМК Пресс, 2010. 232 с.
2. Сандерс Дж., Кэндрот Э. Технология CUDA в примерах: введение в программирование графических процессоров / пер. с англ. А.А. Слинкина, науч. ред. А.В. Боресков. М.: ДМК Пресс, 2011. 232 с.
3. Micikevicus P. 3d finite difference computation on GPUs using CUDA // GPGPU-2 Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units. 2009. P. 79–84.
4. Zhang Y., Cohen J., Owens J.D. Fast Tridiagonal Solvers on the GPU // Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2010. P. 127–136.
5. Egloff D. High perforamnce finite difference PDE solvers on GPUs // Technical report, QuantAlea GmbH, 2010. 28 p.
6. Bell N., Garland M. Efficient sparse matrix-vector multiplication on CUDA // NVIDIA Technical Report, 2008. 32 p.
7. Калиткин Н.Н. Численные методы. М.: Наука, 1978. 512 с.
8. Самарский А.А. Теория разностных схем. М.: Наука, 1983. 616 с.

Материалы компании NVIDIA

1. CUDA C. Getting Started.
2. CUDA C. Programming Guide.
3. CUDA C. Best Practice Guide.
4. NVIDIA Fermi Compute Architecture Whitepaper.
5. CUDA. CUFFT Library User Guide.

6. CUDA. CUBLAS Library User Guide.
7. CUDA. CUSPARSE Library User Guide.
8. CUDA. CURAND Library User Guide.
9. CUDA Toolkit 4.0 Overview.

Информационные ресурсы в сети Интернет

1. Информационные материалы по вычислениям на GPU: http://gpu.parallel.ru
2. Информационные материалы NVIDIA CUDA ZONE: http://developer.nvidia.com/category/zone/cuda-zone/
3. Учебные курсы по параллельным вычислениям: http://www.intuit.ru
4. Информационные материалы по OpenMP: http://www.openmp.org
5. Информационные материалы по MPI: http://www.mpi-forum.org